

Un Web programmable

Œuvre inachevée

Aaron Swartz

UN WEB PROGRAMMABLE

Édition originale

Aaron Swartz's A Programmable Web : An Unfinished Work / Aaron Swartz

Collection : Synthesis Lectures on The Semantic Web: Theory and Technology / James Hendler and YingDing, Series Editors

Morgan & Claypool Publishers

www.morganclaypool.com

ISBN: 9781627051699 ebook

DOI 10.2200/S00481ED1V01Y201302WBE005

Traduction française

Pierre Marige

Licence

Photo en couverture

Aaron Swartz at a Boston Wikipedia Meetup, 2009-08-18 - Sage Ross - CC-BY-SA

Aaron Swartz

Un Web programmable

Œuvre inachevée

À Dan Connolly, qui non seulement a créé le Web, mais a trouvé le temps de me l'enseigner.

Aaron Swartz, Novembre 2009

TABLE

Avant-propos

James Hendler, directeur de la collection « Synthesis Lectures on The Semantic Web : Theory and Technology »

En 2009, j’ai invité Aaron Swartz à contribuer par un « exposé de synthèse » – essentiellement un court livre en ligne – à une nouvelle collection que je dirigeais sur l’ingénierie web. Aaron rédigea un brouillon d’environ 40 pages, le document que vous voyez ici. C’était une « première version » à compléter plus tard. Malheureusement, et à mon grand regret, cela n’eut jamais lieu.

La version fournie par Aaron n’était pas censée être le produit fini. Il l’a écrite relativement vite, et nous l’a envoyée pour commentaires. Je lui en ai envoyé une petite série, et plus tard Frank van Harmelen, qui m’avait rejoint en tant que codirecteur de la collection, lui en envoya une série plus longue. Aaron échangea avec nous un petit peu, mais ensuite passa à d’autres choses et ne put terminer.

À la mort d’Aaron en janvier 2013, nous avons décidé qu’il serait bénéfique de publier cela, pour que les gens puissent lire dans ses propres mots ses idées sur la programmation du Web, son ambivalence à propos de différents aspects de la technologie du Web sémantique, quelques pensées concernant l’ouverture, etc.

Ce document a été produit originellement au format « markdown », un format HTML/Wiki simplifié qu'Aaron avait conçu avec John Gruber autour de 2004. Nous avons utilisé un des nombreux outils markdown disponibles sur le Web pour le convertir en HTML, puis successivement en LateX et en PDF, pour réaliser ce document. Cette version fut également éditée pour corriger quelques erreurs de copier/coller et améliorer sa lisibilité. Une version HTML de l'original est disponible sur http://www.cs.rpi.edu/~hendler/ProgrammableWebSwartz2009.html.

Jim Hendler, Février 2013

Chapitre 1
Introduction : un Web programmable

Si vous êtes comme la plupart des gens que je connais (et, puisque vous lisez ce livre, vous l'êtes probablement – du moins de ce point de vue), vous utilisez le Web. Beaucoup. En fait, dans mon cas personnel, la grande majorité de mes journées sont passées à lire ou survoler des pages web – un balayage le long de mon client webmail pour parler avec amis et collègues, un blog ou deux pour me tenir au courant des infos du jour, une douzaine d'articles courts, une flottille de requêtes Google, et le va-et-vient permanent vers Wikipedia pour une réponse isolée à une question persistante.

Que du bon, bien sûr ; en fait, du presque indispensable. Et ça fait réfléchir quand on pense qu'il y a un peu plus d'une décennie, rien de tout ça n'existait. Les emails avaient leur propres applications spécialisées, les blogs n'avaient pas encore été inventés, les articles se trouvaient sur papier, Google n'était pas né, et Wikipedia même pas une étincelle distante dans l'œil de Larry Sanger.

Du coup, il est saisissant, presque choquant en fait, d'imaginer ce à quoi le monde pourra ressembler quand nos logiciels iront sur le Web aussi fréquemment et aussi

facilement que nous. Aujourd'hui, bien sûr, nous pouvons voir les pâles et futurs miroitements d'un tel monde. Il y a des logiciels qui appellent chez eux pour voir s'il y a une mise à jour. Il y a des logiciels dont une partie du contenu – la page d'aide, peut-être, ou une sorte de catalogue – est diffusée en ligne sur le Web. Il y a des logiciels qui envoient une copie de tout votre travail pour le stocker sur le Web. Il y a des logiciels spécialement conçus pour vous aider à naviguer sur un certain type de page web. Il y a des logiciels qui ne sont rien d'autre qu'un certain type de page web. Il y a des logiciels – qu'on appelle « mashups » – qui consistent en une page web qui combine les informations des deux autres pages web. Et il y a des logiciels qui, par l'intermédiaire d'« APIs », traitent d'autres sites web simplement comme un autre morceau de l'infrastructure du logiciel, une autre fonction qu'ils peuvent appeler pour faire des choses.

Nos ordinateurs sont tellement petits et le Web est tellement grand et vaste que ce dernier scénario semble faire partie d'une tendance irrésistible. Pourquoi ne compteriez-vous pas sur d'autres sites à chaque fois que vous le pouvez, faisant ainsi de leurs informations infinies et de leurs abondantes capacités une partie homogène du vôtre ? Et je pressens que de tels usages deviendront de plus en plus communs jusqu'à ce qu'un jour votre ordinateur soit aussi ligoté au Web que vous l'êtes vous-même aujourd'hui.

Il est parfois suggéré qu'un tel futur est impossible, que fabriquer un Web que d'autres ordinateurs pourraient utiliser est le fantasme de quelques auteurs de science-fiction (à mon avis, sans trop d'imagination). Que cela ne pourrait arriver que dans un monde de robots lourds, d'intelligence artificielle et de machines qui vous suivent en aboyant des ordres tout en, par intermittence, tentant de vous convaincre sans succès d'acheter une nouvelle paire de chaussures.

Il n'est donc peut-être pas si surprenant que l'un des détracteurs qui ont exprimé ce genre de point de vue, Cory Doctorow, soit justement un auteur de science-fiction plutôt imaginatif (entre autres choses). Il expose sa critique dans son essai « Métamerde : mettons le feu à sept épouvantails de la méta-utopie »(1), également édité dans son recueil d'essais « Contenu : essais choisis sur la technologie, la créativité, le droit d'auteur et le futur du futur »(2), lui aussi disponible en ligne (http://craphound.com/content/download/).

Doctorow affirme que tout système tentant de collecter des « métadonnées » précises – le type de données utilisables par les machines qui seront nécessaires pour réaliser ce rêve d'ordinateurs-utilisant-le-web – se heurtera à sept problèmes insurmontables : les gens mentent, les gens sont fainéants, les gens sont bêtes, les gens ne se connaissent pas eux-mêmes, les schémas ne sont pas neutres, les unités de mesure influencent les résultats, et il y a plus d'une façon de décrire quelque chose. À la place, Doctorow propose qu'au lieu d'essayer d'obtenir des gens qu'ils produisent des données, nous devrions regarder les données qu'ils produisent fortuitement en faisant autre chose (comme Google, qui regarde les liens que les gens font quand ils écrivent des pages web), et utiliser plutôt celles-là.

Doctorow, bien sûr, s'attaque à un épouvantail. Le fantasme utopique de données honnêtes, complètes et non-biaisées sur absolument tout est irréalisable. Mais qui tentait ça, en fait ? Le Web est rarement parfaitement honnête, complet et impartial – et pourtant il est sacrément utile. Il n'y a pas de raison qu'on ne puisse pas faire un Web utilisable par les ordinateurs de la même manière.

Je dois dire toutefois que les promoteurs de l'idée sont un peu coupables de ces perceptions utopistes. Beaucoup d'entre eux ont se sont répandus à propos du « Web sémantique » au sein duquel nos ordinateurs seront enfin capables de « compréhension par les machines ». Un tel cadrage (parmi d'autres facteurs) a attiré les réfugiés du monde de l'intelligence artificielle, qui tire le diable par la queue, et ceux-ci ont trouvé là une nouvelle opportunité de promouvoir l'œuvre de leur vie.

Au lieu de l'attitude « construisons simplement quelque chose qui marche » qui a fait du Web (et d'Internet) un tel succès, ils ont amené l'état d'esprit normalisateur des mathématiciens et les structures institutionnelles des universitaires et des fournisseurs de la Défense. Ils ont formé des comités pour former des groupes de travail pour écrire des brouillons d'ontologies (des documents Word de 100 pages) qui listent précisément toutes les choses possibles dans l'univers et les propriétés variées qu'elles peuvent avoir, et ils ont passé des heures en débats talmudiques

1 « Metacrap : Putting the torch to seven straw-men of the meta-utopia » [non-traduit en français]. Disponible en ligne : http://www.well.com/~doctorow/metacrap.htm.

2 « Content: Selected Essays on Technology, Creativity, Copyright, and the Future of the Future » (2008, Tachyon Publications) [non-traduit en français].

sur le fait qu'un lave-vaisselle soit un appareil électroménager de cuisine ou de nettoyage.

Avec eux sont arrivés les recherches universitaires, les subventions gouvernementales, les R&D du privé, et tout l'appareil des gens et des institutions qui ont un projet chimérique. Et au lieu de passer du temps à fabriquer des choses, ils ont convaincu les personnes intéressées par ces idées que le premier besoin était d'écrire des standards (aux yeux des ingénieurs, c'est absurde dès le début – les standards sont ce qu'on écrit après qu'on a obtenu quelque chose qui fonctionne, pas avant !).

Du coup, le groupe « Semantic Web Activity » du World Wide Web Consortium (W3C) a passé son temps à écrire standard sur standard : le langage de balisage extensible (Extensible Markup Language / XML), la structure de description des ressources (Resource Description Framework / RDF), le langage d'ontologie Web (Web Ontology Language / OWL), des outils pour la récolte de descriptions de ressources à partir de dialectes de langages (Gleaning Resource Descriptions from Dialects of Languages / GRDDL), le protocole simple et langage de requête RDF (Simple Protocol And RDF Query Language / SPARQL), créé par le groupe de travail RDF sur l'accès aux données (RDF Data Access Working Group / DAWG).

Peu d'entre eux sont utilisés de manière largement répandue, et ceux qui le sont (XML) sont invariablement les fléaux de la planète, des offenses faites aux programmeurs travailleurs qui ont repoussé des formats sensés (comme JSON) en faveur d'usines à gaz sur-compliquées absolument pas basées sur la réalité (Et je n'en ai pas fini ! – plus là-dessus au chapitre 5).

Au lieu de faire parler des systèmes existants entre eux et de consigner les meilleures pratiques, ces garants auto-proclamés du Web sémantique ont passé leur temps à créer leur propre petit univers, complété par des bases de données en Web sémantique et des langages de programmation. Mais les bases de données et les langages de programmation, même très imparfaits, sont des problèmes largement résolus. Les gens ont déjà leurs préférés, qui ont été testés et bricolés pour fonctionner dans toutes sortes d'environnements inhabituels, et ils ne sont pas particulièrement enclins à en apprendre un nouveau, surtout sans une bonne

raison de le faire. C'est assez difficile de faire en sorte que les gens partagent des données telles quelles, encore plus difficile de les faire les partager dans un format particulier, et complètement impossible d'obtenir qu'ils les conservent et les administrent dans un système totalement nouveau.

Et pourtant c'est à quoi les grosses têtes du Web sémantique passent leur temps. C'est comme si pour amener les gens à utiliser le Web ils avaient commencé par écrire un nouveau système d'exploitation avec le Web directement construit en son cœur. Bien sûr, il y a des chances qu'on en arrive là un jour, mais insister pour que les gens commencent par ça aurait confiné le Web dans l'obscurité dès le premier jour.

Tout cela a conduit les « ingénieurs web » (comme le titre de cette collection les nomme affectueusement) à faire la sourde oreille et à retourner à un vrai travail, parce qu'ils ne voulaient pas perdre leur temps avec des choses qui n'existent pas et qui, fort probablement, n'existeront jamais. Et ça a conduit beaucoup de ceux qui avaient travaillé sur le Web sémantique, dans le vain espoir de réellement construire un monde où les logiciels pourraient communiquer, à craquer, tout plaquer et trouver des voies plus productives auxquelles consacrer leur attention.

Par exemple, voyez Sean B. Palmer. Dans son article remarqué « Laisser tomber le Web sémantique ? »[3], il proclame : « Il n'est pas prudent, et peut-être même pas moral (si cela ne sonne pas trop mélodramatique), de travailler sur RDF, OWL, SPARQL, RIF, l'idée cassée de confiance distribuée, CWM, Tabulator, Dublin Core, FOAF, SIOC, ni sur tout ce genre de choses ». Et, non seulement il va « arrêter de travailler sur le Web sémantique » mais il va « en outre, activement dissuader tout le monde de travailler sur le Web sémantique, s'empêchant ainsi de travailler sur » des projets plus pratiques.

En toute justice, notons bien que je ne suis pas exactement un observateur impartial. D'abord, Sean, à l'instar d'à peu près toutes les personnes que je cite dans ce livre, est un ami. Nous nous sommes rencontrés quand nous travaillions

3 « Ditching the Semantic Web? » [non-traduit en français]. Disponible en ligne : http://inamidst.com/whits/2008/ditching.

ensemble sur ces sujets, et depuis nous avons gardé contact, nous échangeons des emails sur ce sur quoi nous travaillons en ce moment, et nous sommes généralement gentils l'un envers l'autre. C'est le cas pour presque toutes les autres personnes que je cite et critique.

De plus, la raison pour laquelle nous avons travaillé ensemble, c'est que j'ai aussi fait mon temps dans les mines de sel du Web sémantique. Ma première application web était une encyclopédie collaborative, mais ma deuxième, des gros titres agrégés de sites d'information répartis sur le Web, m'a entraîné dans une spirale descendante qui s'acheva, après de nombreuses années passées dans les groupes de travail du noyau RDF par une décision définitive de m'éloigner du monde de l'informatique dans son ensemble.

Manifestement, ça n'a pas marché tout à fait comme prévu. Jim Hendler, un autre ami, et un de ces transfuges de l'Intelligence artificielle sur qui je viens de taper longuement, m'a demandé si je pouvais écrire un petit quelque chose sur le sujet pour entamer une nouvelle collection de livres électroniques qu'il est en train de lancer. Je ferais n'importe quoi pour un peu d'argent (je plaisante : je veux juste être publié (je plaisante : j'ai déjà été publié plein de fois (je plaisante : pas tant de fois que ça (je plaisante : je l'ai été, mais j'ai besoin de m'entraîner (je plaisante : je ne m'entraîne jamais (je plaisante : je voulais juste publier un livre (je plaisante : je voulais juste écrire un livre (je plaisante : c'est facile d'écrire un livre (je plaisante : c'est un chemin de croix (je plaisante : c'est pas si mal (je plaisante : ma copine m'a quitté (je plaisante : je l'ai quittée (je plaisante, je plaisante, je plaisante))))))))))))) et donc me revoilà, reprenant ces vieilles questions, avec enfin une occasion de me plaindre à propos de l'erreur que les gens du Web sémantique avaient faite.

Cependant, comme mon petit exercice de pensée exprimé plus haut l'a je l'espère démontré, le Web programmable est tout sauf une chimère – c'est la réalité d'aujourd'hui et la banalité de demain. Aucun développeur de logiciels ne se contentera de se limiter aux choses présentes sur l'ordinateur de l'utilisateur. Et aucun développeur de site web ne se contentera de limiter son site aux seuls utilisateurs qui interagissent directement avec lui.

Tout comme le pouvoir d'inter-liaison du World Wide Web a aspiré tous les documents disponibles dans son gosier – encourageant les gens à les numériser, à les convertir en HTML, à leur donner une URL, et à les mettre sur Internet (bon sang, pendant que nous parlons Google le fait même pour des bibliothèques entières) – le Web programmable engloutira toutes les applications à portée de main. Les bénéfices du fait d'être connecté sont simplement trop puissants pour qu'on puisse résister.

Bien entendu, ce sera – comme toutes les nouvelles technologies – un défi ouvert pour les modèles économiques, particulièrement pour ceux qui font leur argent sur l'enfermement et l'accès payant aux données. Mais de telles pratiques ne sont tout simplement pas viables sur le long terme, légalement comme d'un point de vue pratique (encore moins moralement). D'après la loi des États-Unis, les faits ne sont pas soumis au droit d'auteur (grâce à la décision de la Cour suprême faisant jurisprudence dans l'affaire Feist v. Rural Telephone Service) et les bases de données sont juste des collections de faits (quelques pays européens ont des droits spéciaux pour les bases de données, mais de telles extensions ont été fermement refusées aux États-Unis).

Et même si la loi ne s'en mêle pas, il est tellement avantageux de partager des données que la plupart des fournisseurs de données vont probablement y venir. Évidemment, fournir un site web où les gens peuvent passer chercher des choses peut être très rentable, mais ça n'est rien en comparaison de ce qui est possible quand on combine ces informations avec d'autres.

Pour prendre un exemple tiré de ma propre carrière, voyez le site web OpenSecrets.org. Il collecte des informations concernant les donateurs aux candidats politiques des États-Unis, et présente de jolis graphiques et tableaux sur les industries qui ont financé les campagnes des candidats à la présidentielle et des membres du Congrès.

De la même manière, le site web Taxpayer.net fournit une profusion d'informations sur les provisions congressionnelles – des demandes de financement que des membres du Congrès glissent dans les projets de lois, réclamant quelques millions de dollars pour quelqu'un pour sa marotte

personnelle (le « pont pour nulle-part » à 398 millions de dollars en étant le plus fameux exemple[(4)]).

Tous les deux sont des sites fantastiques et sont utilisés fréquemment et à profit par les observateurs de la politique américaine. Mais imaginez combien meilleurs ils seraient si vous les réunissiez – vous pourriez chercher les principaux contributeurs à des campagnes qui ont reçus des grosses subventions.

Notez que ce n’est pas le genre de « remix » qu’on peut réaliser avec les APIs d’aujourd’hui. Les APIs permettent seulement de regarder les données d’une certaine façon, généralement de la façon dont le site hébergeur les regarde. Ainsi, avec l’API d’OpenSecrets, on peut obtenir une liste des contributeurs les plus importants d’un candidat. Mais ça n’est pas suffisant pour le type de question qui nous intéresse – il nous faut comparer chaque subvention avec chaque donateur et chercher les concordances. Cela nécessite un accès réel aux données.

Notez aussi qu’en fin de compte, le résultat final est avantageux pour tout le monde. OpenSecrets.org veut que les gens en sachent plus sur l’influence problématique de l’argent sur la politique. Taxpayer.net veut attirer l’attention sur ces dépenses inutiles. Le public veut savoir comment l’argent en politique entraîne des dépenses inutiles, et un site qui les y aiderait servirait l’objectif des deux organisations. Mais elles ne peuvent y parvenir qu’en acceptant de partager leurs données.

Heureusement pour nous, le Web a été conçu avec ce futur à l’esprit. Les protocoles sur lesquels il est fondé ne sont pas simplement conçus pour fournir des pages à la consommation humaine, mais également pour accueillir facilement la ménagerie des araignées, des robots et des scripts qui explorent son sol fertile. Et les développeurs originels du Web, les hommes et les femmes qui ont inventé les outils qui en font le passe-temps chronophage qu’il est aujourd’hui, se sont depuis longtemps intéressés à rendre le Web sûr, et même accueillant, pour les applications.

4 [http://en.wikipedia.org/wiki/Gravina_Island_Bridge].

Malheureusement, trop peu savent cela, et cela en amène beaucoup à réinventer – n'importe comment – le travail qui a déjà été fait (le fait que les rares personnes au courant aient passé leur temps à travailler sur cet absurde Web sémantique que j'ai critiqué plus haut n'a pas aidé). Donc nous commencerons par essayer de comprendre l'architecture du Web – ce qui marche bien et, à l'occasion, ce qui ne marche pas, mais surtout pourquoi il est fait comme ça. Nous allons apprendre comment il autorise à la fois les utilisateurs et les moteurs de recherche à coexister pacifiquement tout en supportant tout, du partage de photos aux transactions financières.

Nous continuerons en réfléchissant à ce que signifie construire un programme au sommet du Web – comment écrire un logiciel qui sert équitablement autant son utilisateur immédiat que les développeurs qui veulent construire par-dessus lui. Trop souvent, une API est boulonnée au-dessus d'une application existante, après-coup ou comme un morceau complètement différent. Mais, comme nous le verrons, quand une application web est conçue proprement, les APIs en découlent naturellement et leur maintenance ne requiert que peu d'efforts.

Puis nous nous intéresserons à ce que signifie pour votre application de n'être pas seulement un autre outil pour les gens et les logiciels, mais une partie de l'écologie – une section du Web programmable. Ce qui implique d'exposer vos données à être interrogées, et copiées, et intégrées, et ce même sans autorisation explicite, au sein du plus grand écosystème de logiciels, tout en protégeant la liberté des utilisateurs.

Enfin, nous conclurons avec une discussion sur cette expression très galvaudée : « Web sémantique », et nous tâcherons de comprendre ce qu'elle signifie vraiment.

Allons-y.

Chapitre 2
Construire pour les utilisateurs : concevoir les URLs

Sur les panneaux d'affichage, les bus, les cartons d'emballage, elles nous épient comme des symboles extra-terrestres (espérons-le, de façon moins menaçante) : les URLs sont partout. De manière plus visible, elles apparaissent en haut de la fenêtre du navigateur quand les gens utilisent votre site web, mais elles apparaissent également dans une myriade d'autres contextes : dans la barre de notifications quand quelqu'un survole un lien avec la souris, dans des résultats de recherche, dans des emails, dans des blogs, lues sur des téléphones, inscrites sur des serviettes de table, listées dans des bibliographies, imprimées sur des cartes de visite et des t-shirts et des tapis de souris et des autocollants de pare-chocs. Ce sont des petits symboles versatiles.

De plus, les URLs doivent tenir. Ces t-shirts, ces liens, ces blogs ne disparaîtront pas parce vous avez décidé de réorganiser votre serveur, ou migré vers un système d'exploitation différent, ou été promu et remplacé par un subordonné (ou pas réélu). Ils tiendront des années et des années, donc vos URLs doivent tenir aussi.

En outre, les URLs ne se contentent pas d'exister en tant qu'entités isolées (comme « http://exemple.org/repas/bacon.html »). Elles se combinent pour former des modèles (« bacon.html », « laitue.html », « tomate.html »). Et chacun de ces modèles trouve sa place sur une plus grande arborescence d'interactions (« / », « /repas », « /repas/bacon.html »).

À cause de tout ça, les URLs ne peuvent pas être des sortes d'effets secondaires ou pensées après-coup, comme certains ont l'air de le souhaiter. Concevoir les URLs est la part la plus importante dans la construction d'une application web, et cela doit être fait en premier. Dans leur conception sont encodées toute une série de présomptions implicites concernant ce dont parle votre site, comment il est structuré, et comment on est censé l'utiliser ; rien que des questions importantes et inévitables.

Malheureusement, de nombreux outils pour fabriquer des applications web essayent de vous cacher ces questions, en vous empêchant complètement de concevoir les URLs. À la place, ils présentent leur propre interface au programmeur, depuis laquelle ils génèrent des URLs en fabriquant des nombres aléatoires, en stockant des cookies, ou pire encore (de nos jours, avec AJAX et Flash, certains ne fournissent plus d'URLs du tout, faisant du site un enfer pour quiconque veut envoyer à ses amis un truc cool qu'il a trouvé).

Et quand les gens qui utilisent ce genre de logiciels se retrouvent à devoir écrire une URL sur un t-shirt, ou faire un lien vers quelque chose depuis un email, ils créent une redirection – une URL spéciale dont l'unique but est d'introduire les gens dans le système cauchemardesque de nombres aléatoires utilisé par leur vrai site web. Cela résout leur problème immédiat de trouver quoi écrire sur le t-shirt, mais ça ne résout aucun des problèmes plus fondamentaux, et ça ne rend pas possible pour quelqu'un d'autre de faire ses propres t-shirts ou d'envoyer ses propres emails.

Si vos outils ne vous permettent pas de concevoir vos URLs, vous avez besoin de meilleurs outils. Personne ne s'aviserait de faire du design graphique avec un logiciel qui ne permettrait pas de changer de police ou qui peindrait avec une brosse qui ne saurait faire que des carrés. Pourtant des gens pensent qu'il est

parfaitement acceptable de sacrifier le contrôle sur les URLs, la part la plus fondamentale de votre site web. Ça ne l'est pas. Procurez-vous de meilleurs outils[5].

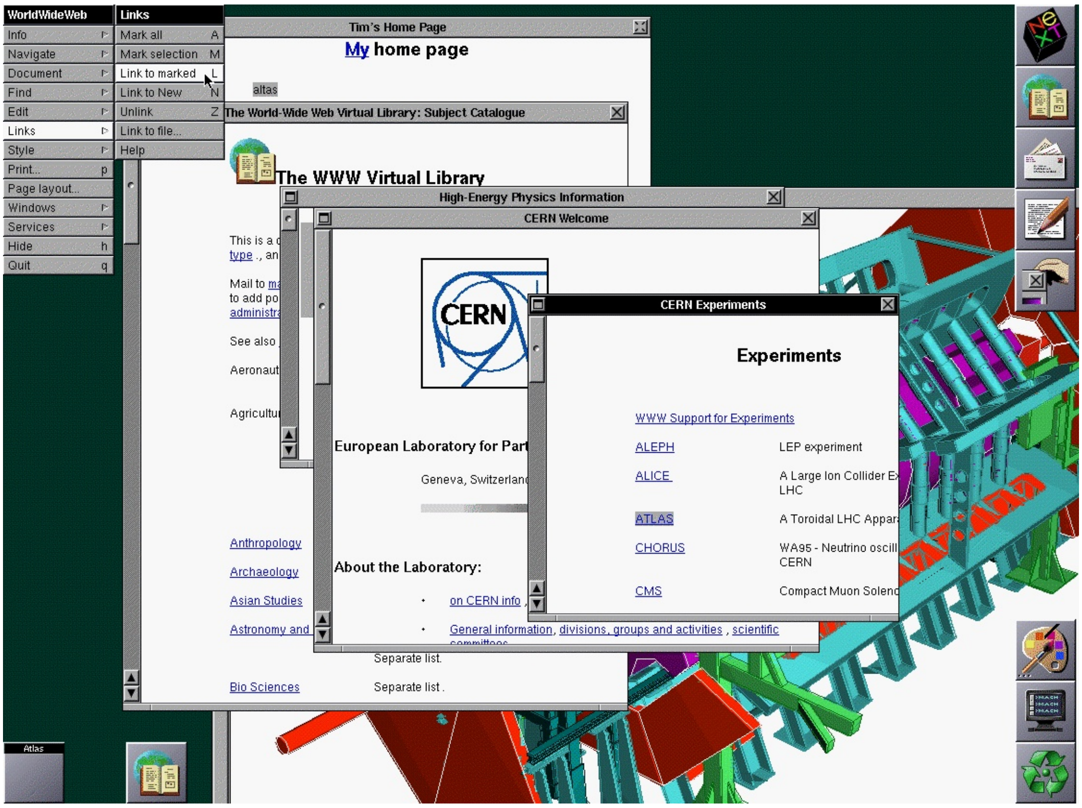
Une fois pourvu d'outils décents, il est temps de commencer à concevoir. Commençons par les plus grandes contraintes. Les URLs ne doivent pas changer (si elles changent, les anciennes doivent rediriger vers les nouvelles), donc elles doivent ne contenir que des informations concernant la page qui ne changent jamais. En découlent quelques impératifs évidents.

Ceux-ci furent indiqués plus notoirement par l'inventeur du Web, Sir Timothy John Berners-Lee, OM, KBE, FRS, FREng, FRSA (né le 8 Juin 1955 à Londres, Angleterre). Pendant un congé de Noël miraculeux, en 1990, qui rappelle une des annus mirabilis d'Einstein, Tim n'a pas seulement inventé les URL, le format HTML, et le protocole HTTP, mais il a aussi écrit le premier navigateur web, le premier éditeur web WYSIWYG, et le premier serveur web (ça donne envie de lui accorder plus de congés de Noël). Bien que, en fait, cela soit un peu redondant, étant donné que le premier navigateur web (appelé WorldWideWeb) non seulement vous permettait de lire des pages web, mais vous permettait aussi de les écrire. L'idée était que le Web soit un média interactif, avec tout le monde qui conserverait son propre carnet de notes de ses trouvailles intéressantes, qui collaborerait sur des documents avec d'autres personnes, et qui posterait des trucs qu'il a fait ou qu'il veut partager.

Éditer une page web était aussi simple que cliquer dessus – vous pouviez simplement basculer en mode édition et sélectionner et corriger les fautes de frappe directement sur la page, comme dans un traitement de texte (vous cliquiez sur enregistrer et ça les versait directement sur le serveur). Vous pouviez créer de nouvelles pages simplement en ouvrant une nouvelle fenêtre, et au lieu de marque-pages, vous étiez censé construire des pages web qui gardent trace des sites que vous trouviez intéressants (le navigateur originel n'avait pas de barre d'adresse, en partie pour vous forcer à marquer vos pages comme cela).

5 Si vous ne savez pas par où commencer, il y a bien entendu mon propre jeu d'outils, web.py (http://webpy.org/), ou encore la plateforme de développement web en Python Django (http://djangoproject.com/).

C'était une idée brillante, mais malheureusement c'était écrit pour NeXT, un obscur système d'exploitation (qui deviendra plus tard Mac OS X), et en conséquence rares sont ceux qui ont pu l'utiliser. À la place, ils ont utilisé le clone créé par une équipe de l'université de l'Illinois Urbana-Champaign (UIUC), qui n'a jamais intégré d'éditeur parce que le programmeur Marc Andreessen était trop bête pour réussir à faire de l'édition avec des images en ligne, ce qui ne posait aucun problème à la version de Tim Berners-Lee. Marc Andreessen a gagné un demi milliard de dollars quand le navigateur de l'UIUC est devenu Netscape, alors que Berners-Lee a continué à faire du support technique pour une équipe de physiciens en Suisse (il deviendra plus tard chercheur au MIT).



Capture d'écran de WorldWideWeb
(http://www.w3.org/History/1994/WWW/Journals/CACM/screensnap2_24c.gif)

Résultat, nous ne nous réapproprions ces merveilleuses caractéristiques qu'une vingtaine d'années plus tard, grâce à des choses comme les blogs ou Wikipedia. Et même là, c'est beaucoup plus limité que l'interactivité de grande envergure que Berners-Lee imaginait.

Mais tournons la page du passé, et revenons au futur. Sir Tim affirmait que pour protéger vos URLs dans le futur, vous deviez suivre quelques principes de base. Dans sa déclaration de 1998 « Les URIs cools ne changent pas »(6), décrite comme « une tentative de rediriger l'énergie consacrée à la quête de coolitude ... vers l'utilité [et] la longévité », il les a exposés.

Cependant, je suis en désaccord avec la solution proposée par Tim pour générer des URIs cools. Il recommande des schémas rigoureusement basés sur la date, comme « http://www.w3.org/1998/12/01/chairs ». D'après ce que j'ai vu, seul le W3C a vraiment adopté rigoureusement cette stratégie, et quand je m'y suis essayé ça n'a donné que laideur et confusion.

Vous remarquez peut-être que Tim parle d'URI alors que je parle d'URL. URL, le terme original, signifie Uniform Resource Locator – Localisateur Uniforme de Ressource. Cela a été développé, en même temps que le Web, pour fournir une façon cohérente de faire référence à des pages web et à d'autres ressources sur Internet. Depuis, cependant, ça a été étendu à une façon de faire référence à toutes sortes de choses, dont la plupart ne sont pas des pages web, et dont certaines ne peuvent même pas être « localisées » de manière automatisée (par exemple, des concept abstraits comme « Time magazine »). En conséquence, le terme a été changé en URI, Uniform Resource Identifier – Identificateur Uniforme de Ressource, pour englober cette sélection plus large. Je m'en tiens ici au terme d'URL, parce qu'il est plus familier, mais nous finirons par évoquer des concepts abstraits dans les chapitres suivants.

Premièrement, vos URLs ne doivent pas inclure de détails techniques à propos du logiciel utilisé pour construire le site web, puisque ceci peut changer à tout moment. Par conséquent, des choses comme « .php » et « .cgi » sont à bannir. Pour

6 Disponible en ligne : http://www.w3.org/Provider/Style/URI.

des raisons similaires, on peut se débarrasser de « .html », de « PHP_SESS_ID », et des choses de ce genre. Vous vous assurerez également que les noms des vos serveurs (par exemple, « www7.exemple.org » ou « platon.exemple.net ») sont absents des URLs. Changer de langage de programmation, de format, ou de serveur est assez courant ; il n'y a pas de raison que vos URLs dépendent de vos choix actuels.

Deuxièmement, on laissera de coté tous les faits concernant la page qui pourraient changer. Ça concerne à peu près tout (son auteur, sa catégorie, qui peut la lire, si c'est officiel ou un brouillon, etc.), les URLs sont donc vraiment limitées au concept essentiel de la page, l'essence de la page. Quelle est la chose qui ne peut pas changer, qui fait que cette page est cette page ?

Troisièmement, vous serez très précautionneux concernant la classification. Beaucoup de gens aiment diviser leur site web par sujets, rangeant leurs recettes favorites dans le répertoire « /nourriture/ » , leurs histoires liées aux voyages qu'ils font dans « /voyages/ », et les trucs qu'ils lisent dans « /livres/ ». Mais inévitablement ils finissent par avoir une recette qui nécessite un voyage ou un livre à propos de nourriture, et ils pensent que ça appartient aux deux catégories. Ou ils décident que les boissons devraient vraiment être extraites et avoir leur propre section. Ou ils décident simplement de tout réorganiser.

Quelle que soit la raison, ils finissent par réarranger leurs fichiers et changer leur structure de répertoires, en cassant toutes leurs URLs. Même si vous réarrangez simplement l'apparence du site, cela demande beaucoup de discipline de ne pas déplacer les fichiers, plus sans doute que vous n'en aurez. Et mettre au point des redirections pour tout est tellement difficile que vous n'allez même pas vous en préoccuper.

Le mieux, c'est de faire en sorte en amont de ne jamais devoir être confronté à ce problème en laissant les catégories en dehors des URLs.

Ça fait beaucoup d'interdictions, que diriez-vous de quelques recommandations ?

Et bien, une façon facile d'obtenir des URLs sûres, c'est de prendre des nombres.

Ainsi, par exemple, votre système de blog pourrait simplement assigner à chaque article un identifiant séquentiel, et leurs donner des URLs comme :

```
http://www.posterous.com/p/234

http://www.posterous.com/p/235

http://www.posterous.com/p/236
```

Rien de mauvais là-dedans. Cependant, si votre site est un peu plus populaire, les identifiants peuvent devenir plutôt longs et déroutants :

```
http://livres.exemple.org/b/30283833
```

Dans une situation comme celle-ci, vous pouvez encoder les nombres en base 36 au lieu d'en base 10. En base 36, on utilise toutes les lettres en plus des chiffres, mais dans une seule casse, pour qu'il n'y ait pas de confusion concernant le passage en majuscules (imaginez quelqu'un lire l'URL au téléphone ; c'est beaucoup plus simple de dire « gé, cinq, enne, quatre » que « gé minuscule, le chiffre cinq, enne majuscule, le chiffre quatre »).

Pour être super prudent, vous pouvez allez plus loin et sauter tous les nombres qui comportent zéro, O, un, L, ou I, étant donné que ces caractères peuvent souvent être confondus.

Vous finissez avec des URLs qui ressemblent à ça :

```
http://livres.exemple.org/b/3j7is
```

et qui sont beaucoup plus courtes. Alors que quatre caractères en base 10 ne peuvent aller que jusqu'à 9999, en base 36 zzzz vaut 1 679 615. Pas mal.

Un problème des identifiants numériques, cependant, c'est qu'ils ne sont pas « optimisés » pour les moteurs de recherche. Les moteurs de recherche ne

regardent pas seulement le contenu de la page pour décider si c'est un bon résultat pour la recherche de quelqu'un, ils regardent aussi l'URL à qui, parce qu'elle est très limitée, ils donnent un poids spécial. Mais si vos URLs sont simplement des nombres, il est peu probable qu'elles correspondent à la requête de quelqu'un sur un moteur de recherche, et ce qui réduit les chances qu'elles soient trouvées dans les résultats de recherche. Pour remédier à cela, les gens ajoutent du texte après le nombre, comme dans :

```
http://www.hulu.com/watch/17003/saturday-night-live-
weekend-update-judy-grimes
```

Le texte à la fin fait partie de l'URL, mais il n'est pas utilisé pour identifier la page. À la place, le système regarde seulement le nombre. Un fois qu'il l'a récupéré, il regarde le titre correspondant au nombre, vérifie s'il correspond à l'URL, et si ce n'est pas le cas, il redirige les utilisateurs vers la bonne URL. De cette manière, ils peuvent saisir :

```
http://www.hulu.com/watch/17003/
```

voire même :

```
http://www.hulu.com/watch/17003/c-est-la-que-j-ai-vu-la-
blague-citee-plus-haut
```
[7]

et quand même atterrir sur la bonne page. Ca n'est pas parfait, puisque beaucoup d'utilisateurs vont penser qu'ils doivent saisir le long texte « saturday-night-live-weekend-update-judy-grimes », mais c'est probablement contre-balancé par le

7 Il est intéressant que, même si nous lisons habituellement des livres comme des séries de bouts de papiers rangées de gauche à droite, on fasse encore référence aux choses mentionnées plus tôt comme si elles était « plus haut » que les autres (voire « supra », si vous préférez le Latin), comme si on lisait tous encore le rouleau brut que Kerouac a sorti de sa machine à écrire (pour plus de détails, voir par exemple :
http://www.npr.org/templates/story/story.php?storyId=11709924).
Bien entendu, contrairement à mes prédécesseurs à machine à écrire ou à carnet de notes relié, j'écris ceci dans un logiciel de traitement de texte dont l'orientation haut-bas simulée imite parfaitement le rouleau matériel de Kerouac. Je suppose que tout cela est cyclique.

nombre d'utilisateurs additionnels qui vont vous trouver plus facilement sur les moteurs de recherche (dans l'idéal, il pourrait y avoir une façon dans l'URL d'indiquer aux humains que le texte supplémentaire est optionnel, mais je n'ai pas encore vu de convention là-dessus ; j'imagine qu'on espère qu'ils vont remarquer le nombre, et comprendre l'idée).

(Vous noterez que toutes ces URLs sont dans des répertoires, et pas au niveau le plus haut. Ça me parait plus propre – je n'aime pas l'idée d'un site dont tous les fichiers seraient étalés aléatoirement sous le répertoire racine ; c'est bien plus agréable d'imaginer tous les fichiers du site empilés dans « /watch/ » ou « /b/ ». Mais si vos noms principaux sont eux-mêmes des sous-répertoires, comme pour les pages d'utilisateurs sur Twitter ou Delicious, cela peut faire sens d'enfreindre cette règle. On y reviendra bientôt.)

Les nombres fonctionnent bien dans les cas où les pages sont créées automatiquement (parce que vous importez beaucoup de trucs, ou vous générez des pages en réponse à des emails, ou automatiquement pour d'autres raisons), ou quand leurs titres ont tendance à changer, mais dans d'autres cas, vous pouvez préférer ce qu'on appelle un « slug ». Un slug est simplement un petit bout de texte qui rend bien dans une URL, comme « wrt_dfw » ou « beyond-flash ». Quand un utilisateur crée une page, il crée le slug en même temps (peut-être avec un slug auto-généré par défaut à partir du titre), et ensuite vous le forcez à le conserver (ou alors, assurez-vous de rediriger tous les anciens s'il en change).

Sur les sites comme Wikipedia, les slugs sont générés incidemment. Quand vous insérez un texte comme « Jackson n'était guère admirateur des dernières œuvres de [[Robert Davidson]] », le site crée automatiquement un lien vers une nouvelle page avec le slug « Robert_Davidson ». Grâce particulièrement aux nombreuses conventions sur les titres que Wikipedia a établies au cours des années (et grâce en même temps aux redirections de sauvegarde infinies), le résultat est étonnamment pratique.

Vous remarquerez que toute cette discussion n'a concerné, au fond, que des noms – les choses qui composent principalement votre site, quelles qu'elles soient (vidéos, articles de blog, livres). Il y a généralement trois autres types de pages :

les sous-pages (qui creusent un certain aspect du nom), les pages du site (comme « à propos », l'aide, et tout ça), et les verbes (qui vous permettent de faire des choses avec les noms.

Les sous-pages sont à la fois les plus faciles et les plus compliquées. Dans les cas simples, vous indiquez simplement la sous-page en ajoutant un slash et un slug pour la sous-page. Donc si votre page sur Nancy Pelosi est à l'adresse :

```
http://watchdog.net/p/nancy_pelosi
```

il semble plutôt évident que votre page concernant ses finances devrait être à l'adresse :

```
http://watchdog.net/p/nancy_pelosi/finances
```

Parfois, la majorité de votre site est composée de sous-pages. Ainsi sur Twitter une page d'utilisateur est à l'adresse :

```
http://twitter.com/aaronsw
```

tandis que leurs messages de statuts ont des URLs de type :

```
http://twitter.com/aaronsw/statuses/918239758
```

(On notera que le morceau « statuses » est redondant, et que le nombre est beaucoup trop long.)

Mais les choses se compliquent quand vos noms ont entre eux des relations plus complexes. Prenez Delicious, où les utilisateurs postent des liens sous divers tags. Comment cela devrait-il être structuré ? user/link/tag ? tag/user/link ?

Delicious, qui pendant longtemps a utilisé son schéma d'URLs comme principale interface de navigation, est tellement brillant dans ses choix d'URLs que son étude mérite toute notre attention. Ils ont décidé que les utilisateurs étaient l'objet

principal, et leur ont donné toute la place dans « / » (comme Twitter). Et sous chaque utilisateur, on peut filtrer par tags, on a donc :

`http://delicious.com/aaronsw` (mes liens)

`http://delicious.com/aaronsw/video` (parmi mes liens, ceux tagués « video »)

`http://delicious.com/aaronsw/video+tech` (parmi mes liens, ceux tagués « video » et « tech »)

Et ils ont créé un pseudo-utilisateur spécial, appelé tag, qui vous permet de voir tous les liens avec un tag :

`http://delicious.com/tag/tech` (tous les liens tagués « tech »)

(Les URLs des liens ne sont pas si malines, mais passons là-dessus.) Il est difficile de donner des règles générales sur comment résoudre ces problèmes d'inter-liaison ; en gros, vous devez faire ce qui « colle bien » avec votre application. Pour les sites sociaux, comme Delicious et Twitter, cela signifie mettre l'accent sur les utilisateurs, puisque c'est principalement ce qui intéresse les utilisateurs. Mais pour d'autres applications ça peut être moins évident.

Il est tentant de tout simplement ne pas décider, et de tout autoriser. Ainsi, à la place de Delicious, on aurait :

`http://del.exemple.org/u/aaronsw` (mes liens)

`http://del.exemple.org/t/tech` (tous les liens tagués « tech »)

`http://del.exemple.org/u/aaronsw/t:tech` (parmi mes liens, ceux tagués « tech »)

`http://del.exemple.org/t/tech/u:aaronsw` (les liens tagués « tech » parmi mes liens)

Le problème ici est que les deux derniers sont des doublons. Il faut vraiment choisir une forme et s'y tenir, sinon on finit par perturber les moteurs de recherche, les historiques de navigateur, et tous les autres outils qui essayent de garder une trace de s'ils ont visité une page ou non. Si vraiment vous avez plusieurs façons de parvenir à la même page, vous devriez en choisir une comme étant l'officielle, et vous assurer que toutes les autres sont des redirections (pour pousser à l'extrême, vous prendriez l'exemple « video+tech » d'au-dessus, et le redirigeriez vers « tech+video », en faisant de l'URL où les tags sont classés par ordre alphabétique l'URL officielle).

Ensuite, les pages de site. En regardant plus haut comment Twitter et Delicious, en gros, donnent les clés du magasin (vous voulez dire que je peux avoir « twitter.com/contact » si mon nom d'utilisateur est « contact » ?!), vous vous demandez peut-être où ils peuvent bien mettre leurs pages d'aide ou de connexion. Une astuce serait de réserver un sous-répertoire comme « /meta/ », et de tout mettre là-dedans. Mais il semble que Delicious et Twitter s'en soient sortis simplement en réservant tous les noms de pages potentiels importants, et ils ont mis leurs affaires dedans. Donc, comme vous vous en doutez, la page de connexion est à l'adresse :

```
http://twitter.com/login
```

Si vous ne vous attendez pas à avoir beaucoup de pages de site, ça devrait vous aller (assurez-vous, malgré tout, de réserver « help »/« aide » et « about »/« a-propos »).

Bien sûr, si vous ne donnez pas les clés du magasin, vous n'avez aucun de ces problèmes. Choisissez juste des URLs sensées pour les pages auxquelles les utilisateurs s'attendent. Et, bien entendu, assurez-vous de suivre tous les principes liés aux noms listés plus haut.

C'était facile, il ne nous reste donc plus que les verbes. On peut imaginer pour les verbes deux façons de fonctionner :

on donne le nom au verbe : `/share?v=1234`

on donne le verbe au nom : `/v/1234?m=share`

Après avoir passé beaucoup de temps à expérimenter là-dessus, je suis convaincu que la deuxième méthode est la bonne. Elle prend moins de place dans « l'espace URL », c'est plus joli dans la barre d'adresse, et elle rend visuellement clair que l'on fait quelque chose à un objet.

Il est tentant d'utiliser des sous-pages, comme :

```
/v/1234/share
```

mais je préfère la formulation « ?m=share » pour deux raisons : d'abord, ça marche même quand le nom a déjà des sous-pages, et ensuite, ça rend clair que la page est censée faire quelque chose, et non simplement apporter plus d'informations. Mais la réciproque est vraie. Ne faites pas :

```
/p/nancy_pelosi?m=finances
```

qui donne l'impression que la page est supposée faire quelque chose alors qu'elle se contente en réalité d'apporter plus d'informations.

Très bien, assez parlé de choisir des URLs. Faisons maintenant quelque chose avec elles !

Chapitre 3

Construire pour les moteurs de recherche : respecter REST

Parlons un peu d’aspirateurs. C’est une histoire des plus banales. Vous avez un joli appartement resplendissant, mais il ne reste pas comme ça longtemps. La poussière tombe sur le sol, les miettes s’échappent de votre assiette, les débris, les épaves, et les petites pièces des figurines des Jetsons commencent à encombrer le chemin. Il est temps de nettoyer.

Balayer, c’est drôle au début – ça vous donne un peu de temps pour vous perdre dans vos pensées à propos de votre application web tout en vous livrant à une activité répétitive et ostensiblement utile – mais on s’en fatigue vite. Pourtant, une culpabilité de gauche et ces articles de Barbara Ehrenreich que vous avez lus vous font renâcler à embaucher une femme de ménage. Donc au lieu d’importer d’un pays étranger une femme fauchée pour faire votre ménage, vous embauchez un robot.

Seulement, il y a cette chose avec les robots (et certaines femmes de ménage, à ce point de vue) : la distinction n’est pas claire du tout pour eux entre ce qui est un déchet et ce qui a de la valeur. Ils (les robots) se baladent dans la maison en

essayant d'aspirer des choses, mais sur leur chemin ils peuvent laisser des traces de roues sur votre manuscrit, renverser votre vase précieux, ou aspirer votre collection de pièces de monnaies antiques. Et parfois, il se prend dans le cordon des stores, et le robot tourne en rond tout en ouvrant sur les volets en tirant dessus.

Alors vous prenez des précautions – avant de lancer le robot, vous relevez le cordon au-dessus du sol, posez votre manuscrit sur le bureau, et faites attention à ne pas laisser votre pile de pièces rares dans le coin. Vous vous assurez que l'endroit est prêt pour que le robot puisse faire son travail sans causer de réel dommage.

C'est exactement la même chose sur le Web (à l'exception de la poussière, des miettes, des figurines Jetsons, des femmes de ménage, des traces de roues, des vases, des pièces et des stores). Les robots (principalement ceux des moteurs de recherche, mais d'autres viennent des spammeurs, de lecteurs déconnectés, et de dieu sait quoi d'autre) parcourent toujours votre site, sans laisser un coin ou une fente inexploré, aspirant tout ce qu'ils peuvent trouver. Et contrairement à la variété éléctroménagère, vous ne pouvez pas vous contenter de les débrancher – vous devez absolument vous assurer de garder les choses propres[8].

Certaines personnes pensent qu'elles peuvent simplement bloquer les robots dehors. « Oh, il faut s'identifier pour entrer ; ça va retenir les robots ». C'est ce que disait David Heinemeier-Hansson, le créateur de Rails. Il se trompait. Les logiciels de Google qui tournaient sur les ordinateurs des usagers ont fini par explorer même les pages qui nécessitaient une identification, ce qui signifie que les robots ont cliqué sur tous les liens « Supprimer », ce qui signifie que les robots ont effacé tout le contenu (Hansson, pour sa part, a répondu en pleurnichant contre l'injustice de tout ça). Ne laissez pas cela vous arriver.

Par chance, ce génial Tim Berners-Lee (voir le chapitre précédent) avait anticipé cela et pris des précautions. Vous voyez, quand vous visitez un site web, vous ne demandez pas seulement l'URL au serveur, vous lui dites aussi quel type de requête vous faites. Voici une requête HTTP/1.0 typique :

8 Voir http://ftrain.com/robot_exclusion_protocol.html.

```
GET/about/HTTP/1.0
```

La première partie (« GET ») s'appelle la méthode, la deuxième (« /about/ ») est le chemin, et la troisième (« HTTP/1.0 ») est évidemment la version. GET est la méthode avec laquelle nous sommes sans doute tous familiers – c'est la méthode normale qu'on utilise pour obtenir (to get, en anglais) une page. Mais il y a aussi une autre méthode : POST.

Si vous pensez les URLs comme des petits programmes installés quelque part dans un serveur, on peut considérer que GET fait juste tourner le programme et récupère une copie de ce qu'il en sort, alors que POST lui envoie plutôt un message. En fait, à la différence des requêtes GET, les requêtes POST viennent avec un chargement. Un message est accroché dessus, pour que l'URL en fasse ce qu'elle veut.

Ça sert pour les requêtes qui font quelque chose qui bouleverse l'ordre de l'univers (ou, dans le jargon, qui « changent l'état »), au lieu de simplement essayer de comprendre qu'est-ce qui est quoi. Ainsi, par exemple, lire une vieille histoire est un GET, puisque vous essayez simplement de comprendre des trucs, mais ajouter un article à votre blog est un POST, puisque vous changer réellement l'état de votre blog.

(Maintenant, si vous voulez jouer au con, vous pouvez dire que toutes les requêtes bouleversent l'ordre de l'univers. Chaque fois que vous demandez une vieille histoire, ça utilise de l'électricité, ça déplace des têtes sur des disques durs, ça ajoute une ligne sur le journal du serveur, ça met une note dans votre dossier à la NSA, etc. C'est absolument vrai, mais il semble bien que ça n'a rien à voir avec ce dont on parle, alors n'en parlons plus. Pardon, la NSA ?)

Le résultat final est très clair. Ça va si Google parcourt et lit de vieilles histoires, mais ça ne va pas s'il vient poster sur votre blog (ou pire, y effacer des choses). Ce qui implique que lire des histoires doit être du GET, et effacer des articles de blog du POST.

En réalité, ça n'est pas tout à fait vrai. Il y a d'autres verbes au coté de GET et POST (bien que ceux-ci soient, de loin, les plus courants). Il y a GET, HEAD,

POST, PUT, DELETE, CONNECT, OPTIONS, PATCH, PROPFIND, PROPPATCH, MKCOL, COPY, MOVE, LOCK, UNLOCK, TRACE (et probablement d’autres encore). GET et POST, nous les avons déjà vus. HEAD est comme GET, mais ne réclame que l’en-tête de la page et pas son véritable contenu. PUT est là si vous voulez changer le contenu de la page par quelque chose d’entièrement nouveau – le navigateur web originel de Tim Berners-Lee utilisait PUT quand vous tentiez de sauvegarder un changement fait sur la page. PATCH est comme PUT, mais ne change qu’une partie de la page. DELETE (supprimer), MOVE (déplacer), COPY (copier), LOCK (verrouiller), et UNLOCK (déverrouiller) s’expliquent plutôt bien d’eux-mêmes. CONNECT s’utilise pour proxyfier et « tunneler » d’autres choses. OPTIONS vous laisse découvrir ce que tolère le serveur. PROPFIND et PROPPATCH sont utilisés pour régler les propriétés dans le protocole WebDAV. MKCOL sert à faire des collections WebDAV (ces derniers n’auraient sans doute pas dû avoir leurs propres méthodes). TRACE demande au serveur de simplement répéter la requête qu’il a reçue (c’est utile pour le débogage).

Mais bon, franchement, GET et POST sont les plus fréquents, en grande partie parce que ce sont ceux supportés par tous les navigateurs web. GET, bien sûr, et utilisé à chaque fois que vous saisissez une URL ou cliquez sur un lien, tandis que POST peut être utilisé dans certains formulaires (les autres formulaires sont en GET, donc ça ne change rien).

Suivre ces règles, c’est respecter REST, d’après la thèse de doctorat de 2000 de Roy Fielding, coauteur (avec Tim Berners-Lee et quelques autres) des spécifications officielles du HTTP (RFC 2616, si ça vous intéresse). Roy, une sorte d’ours avec un penchant pour le sport, s’est proposé de décrire théoriquement les divers styles (« architectures ») d’applications basées sur le réseau. Puis il a décrit l’hybride intéressant que le Web a adopté, qu’il appelle « transfert d’état représentationnel » (Representational State Transfer) ou REST.

Bien que REST soit souvent évoqué pour dire quelque chose comme « utiliser GET et POST correctement », c’est en réalité bien plus complexe et intéressant que ça, et nous allons passer un peu de temps dessus pour que vous puissiez voir les différentes sortes de compromis architecturaux auxquels ont dû penser ces maîtres de l’univers qui ont conçu un système comme le Web.

Le premier choix fût que le Web serait un système client-serveur. Honnêtement, le Web est probablement comme ça parce que Tim a fait les choses comme ça, et Tim a fait les choses comme ça parce que c'est comme ça que toutes les choses sur Internet étaient à l'époque. Mais il n'est pas impossible d'imaginer que le Web aurait pu être plus pair-à-pair, comme certains services d'échange de fichiers qu'on voit aujourd'hui (après tout, le Web est en grande part du simple échange de fichiers).

L'option la plus facile serait, bien sûr, de rompre avec le Web dans son ensemble, et de forcer les gens à télécharger un logiciel particulier pour utiliser votre application. Après tout, c'est comme ça que fonctionnaient la plupart des applications avant le Web (et que fonctionnent encore beaucoup aujourd'hui) – nouveau logiciel, nouveaux protocoles, nouvelle architecture pour chaque application. Il y a certainement de bonnes raisons de faire cela, mais ça vous coupe du reste de la communauté du Web – on ne peut pas faire de lien vers vous, vous ne pouvez pas être parcouru par Google, vous ne pouvez pas être traduit par Babelfish, etc. Si c'était un choix que vous souhaitiez faire, vous ne seriez probablement pas en train de lire ce livre.

Le deuxième choix majeur fût que le web serait « sans état ». Imaginez une connexion réseau où votre ordinateur appellerait le QG et commencerait une conversation. Dans un protocole avec états, il y a de longues conversations – « Allô ? » « Allô, bienvenue sur Amazon. Je suis Shirley. » « Bonjour Shirley, comment-allez-vous ? » « Oh, très bien, et vous ? » « Oh, bien, très très bien. » « J'en suis ravie. Que puis-je faire pour vous ? » « Eh bien, je me demandais ce que vous aviez dans votre rayon littérature. » « Hmm, laissez-moi voir. Ah, il semble que nous ayons plus de 15 millions de livres. Pourriez-vous être un peu plus précis ? » « Oui, est-ce que vous en avez de Dostoïevski ? » (etc.). Mais le Web est sans état – chaque connexion commence à zéro, sans antériorité.

Ça a ses bons cotés. Par exemple, si vous êtes en pleine recherche d'un livre sur Amazon quand, juste au moment où vous êtes sur le point de le trouver vous regardez l'heure et saperlipopette ! il est tard, vous allez manquer votre avion ! Alors vous claquez votre ordinateur portable, vous le fourrez dans votre sac, vous vous ruez vers la porte, vous embarquez dans l'avion, et finalement, vous arrivez à l'hôtel après une journée entière, et rien ne vous empêche de rouvrir votre portable

dans ce pays complètement différent et de reprendre votre recherche là où vous en étiez resté. Tous les liens fonctionnent toujours, après tout. Une conversation avec état, en revanche, ne survivrait certainement pas à une pause d'un jour ou à un changement de pays (de la même manière, vous pouvez envoyer un lien vers votre recherche à un ami de l'autre coté du monde et vous deux pourrez l'utiliser sans problème).

C'est bénéfique pour le serveur également. Au lieu d'avoir chaque client accroché à une partie d'un serveur en particulier pour aussi longtemps que durera la conversation, les conversations sans état sont expédiées très rapidement, et peuvent être tenues par n'importe quel vieux serveur, étant donné qu'ils n'ont pas besoin de connaître un historique.

Certaines mauvaises applications web essayent de s'affranchir de la nature sans état du Web. La façon la plus courante de le faire est l'utilisation de cookies de session. Maintenant, il y a assurément de bonnes raisons d'utiliser des cookies. Exactement comme quand vous appelez votre banque et qu'ils vous demandent votre numéro de compte pour pouvoir retrouver votre dossier, les cookies peuvent permettre aux serveurs de construire des pages arrangées sur-mesure pour vous. Il n'y a rien de mal avec ça.

(Cependant on pourrait se demander si les utilisateurs ne gagneraient pas à utiliser la méthode d'authentification Digest, plus sûre et construite en HTTP, mais comme à peu près toutes les applications sur le Web utilisent aujourd'hui des cookies, c'est sans doute une cause perdue. Il y a quelques espoirs d'amélioration en HTML5 (la prochaine version d'HTML) puisqu'ils – oh, attendez, ils ne vont pas s'occuper de ça. Hmm, bon, je vais tenter de le leur suggérer.)(9)

Le vrai problème apparaît quand on utilise les cookies pour créer des sessions. Par exemple, imaginez si Amazon.com n'avait qu'une URL : http://www.amazon.com/. La première fois que vous visiteriez le site, il vous donnerait la page d'accueil et un numéro de session (disons 349382). Ensuite, vous enverriez un rappel en disant « Je suis la session numéro 349382 et je veux

9 http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2008-October/016742.html.

chercher des livres », et il vous renverrait en retour la page des livres. Puis vous renverriez un rappel en disant « Je suis la session numéro 349382 et je veux chercher Dostoïevski ». Et ainsi de suite.

Aussi fou que cela puisse paraître, beaucoup de sites fonctionnent de cette façon (et encore plus l'ont fait par le passé). Pendant des années, le plus coupable fût probablement la boite à outils appelée WebObjects, célèbre particulièrement pour avoir piloté le magasin en ligne d'Apple. Mais, après des années et des années, il semblerait que WebObjects ait été réparé. Pourtant, de nouvelles plateformes comme Arc et Seaside surgissent pour prendre sa place. Toutes font ça pour la même raison basique : ce sont des logiciels pour construire des applications web qui veulent garder le Web hors de votre vue. Ils veulent faire ça pour que puissiez écrire un logiciel normalement, et qu'il devienne magiquement une application web, sans que vous ayez à fournir le travail de penser les URLs ou de respecter REST. Eh bien, vous pouvez obtenir à partir de ça une application utilisable avec un navigateur, mais vous n'obtiendrez pas une application web.

Le morceau suivant de l'architecture du Web est la mémoire cache. Puisque nous avons cette longue série de requêtes sans état, ça serait sûrement bien si on pouvait les garder en cache. Est-ce que ça ne serait pas formidable si chaque fois que l'on presse le bouton retour, le navigateur n'avait pas besoin de retourner au serveur et de recharger toute la page ? Si, bien sûr. C'est pourquoi tous les navigateurs enregistrent les pages en cache – ils en garde une copie en local et se contentent de vous la présenter si vous en refaites la requête.

Mais il n'y a pas de raison de se limiter au cache des navigateurs. Les FAI aussi utilisent parfois la mémoire cache. Ainsi, si une personne télécharge la bande-annonce du dernier gros film, le FAI peut en garder une copie, et servir le même fichier à tous leurs clients. Ça rend les choses bien plus rapides pour les clients (qui ne sont pas en compétition avec le monde entier pour les mêmes fichiers) et bien plus faciles pour l'opérateur du serveur (qui n'a plus besoin de fournir autant de copies). Le seul problème est que ça a tendance à perturber un peu vos statistiques de téléchargement, mais les opérateurs de serveurs peuvent décider si le jeu en vaut la chandelle.

De la même façon, les serveurs peuvent utiliser la mémoire cache. Plutôt que les navigateurs visitent directement le serveur, ils tombent sur un serveur cache (appelé techniquement un reverse proxy, ou proxy inverse) qui vérifie s'il ne possède pas déjà une copie de la page et qui, s'il l'a, la fournit, et sinon la réclame au véritable serveur. Si vous construisez votre application web en respectant REST, vous pouvez souvent rendre votre site beaucoup, beaucoup plus rapide en collant simplement un chouette serveur cache (comme Polipo) devant lui. Mais, évidemment, si vous faites mal les choses en utilisant par exemple des cookies de sessions et en ignorant les règles de GET et POST, le serveur cache va juste tout bousiller (notez que seules les requêtes GET peuvent être sauvées en cache ; vous ne voudriez pas mettre en cache le résultat de quelque chose comme ajouter un nouvel article de blog, sinon l'article suivant ne serait jamais ajouté !).

GET et POST sont, bien sûr, une partie du morceau suivant de l'architecture que Fielding appelle « interfaces uniformes ». Toutes les applications web fonctionnent de la même manière : elles sont une série d'URLs sur lesquelles vous appliquez des méthodes. Les méthodes changent parfois l'état de l'objet, et le serveur renvoie toujours la « représentation » de l'objet qui en résulte.

D'où son nom : transfert d'état représentationnel, Representational State Transfer, REST.

CHAPITRE 4

CONSTRUIRE POUR LE CHOIX : AUTORISER IMPORTATION ET EXPORTATION

Les robots, les navigateurs et les protocoles sont sympas, c'est sûr, mais si vous voulez que votre site ait du succès, il faut qu'il plaise aux humains – les vrais gens qui construisent et utilisent tous ces autres trucs. Et même si ça n'est pas le cas de l'information, les humains veulent habituellement être libres. Si vous ne me croyez pas, demandez à un ami de vous enfermer dans son coffre de voiture, puis réévaluez votre position.

Les gens cupides (i.e les hommes d'affaires) ont tendance à ne pas voir plus loin que le bout de leur nez dans ce domaine. « Si je mets des grosses pointes de métal devant la sortie », pensent-ils, « mes clients ne voudront jamais partir ! Mon taux de rétention de clientèle va monter en flèche ». Ils décident d'essayer, et font installer quelques grosses pointes de métal devant la sortie. Et, comme les hommes d'affaires aiment prétendre qu'ils sont des personnes réalistes et sensées, ils mesurent les taux de rétention de clientèle avant et après l'installation des pointes de métal. Et, bien évidemment, ça a fonctionné – les gens ne partent pas. Jetez juste un œil à ces chiffres ! Mais ce qu'ils ne mesurent pas, c'est que les gens ne reviennent pas non plus. Après tout, personne ne veut aller quelque part où il y

a des pointes à la sortie. Pensez-y la prochaine fois que vous trouvez que les pubs en pop-up augmentent le taux de rotation du stock.

C'est pourquoi un site comme Amazon est un tel fouillis d'encarts de vente. Les gestionnaires d'Amazon insistent sur le fait qu'ils sont des ingénieurs rigoureusement réalistes. Les encarts sont là parce qu'ils vendent des choses et leur rôle est de rapporter de l'argent. Les pages propres, claires, sans désordre plaisent peut-être aux gamins en école d'art ou aux stagiaires de chez Apple, mais ici dans le monde réel, l'argent fait loi. Et, à l'instar de Mark Penn conseillant Hillary Clinton, si vous ne les croyez pas, ils sortent les chiffres pour le « prouver ». Chaque encart, disent-ils, a été soigneusement testé : on a donné à la moitié des utilisateurs une page avec le nouvel encart, et à l'autre moitié une page sans. Et les utilisateurs à qui on a donné la page avec l'encart ont plus acheté.

Oui, sans doute. Manifestement, il y a plus de gens qui vont acheter quelque chose si on leur propose, tout comme plus de clients de McDonald vont choisir le grand menu quand l'adolescent bougon va le leur proposer. Mais Amazon va plus loin que ça – là, on est dans le royaume de la fille-au-micro-casque qui nous relance tous les trois articles du menu. C'est sûr, on va acheter plus cette fois, mais après l'indigestion on se promettra que la prochaine sortie se fera au Burger King.

Les entreprises essayent rarement de mesurer ce genre d'effets et même quand elles le font, ça n'est pas facile. C'est enfantin de donner à quelqu'un un lien additionnel, et de regarder s'il clique dessus ; garder trace de leur éventuel retour au magasin dans les semaines ou les mois à venir est bien plus difficile. Pire encore, la différence faites par un encart en plus est subtile, et donc compliquée à mesurer. Le vrai test n'est pas de vérifier si Amazon attire plus de clients en retirant un encart, mais plutôt en optant pour une apparence plus douce et agréable. Mais ça serait un changement radical pour Amazon – et donc très difficile à tester sans en hérisser ou effrayer certains. « Eh bien, si on ne peut pas mesurer les gens », disent les MBAs, « on peut au moins leur demander ». D'où les redoutables « focus groups », dont les défauts peuvent éclipser même la plus fausse des études statistiques. Au moins, en jouant avec les clics, vous mesurer ce que les gens font vraiment ; avec les focus groups vous découvrez ce que les gens veulent que vous pensiez qu'ils disent qu'ils font, ce qui est très différent.

Déjà, les gens sont notoirement de mauvais observateurs d'eux-mêmes. Pour la plupart, on ne sait pas pourquoi ni comment on fait les choses, donc quand on nous demande on improvise sur le coup une rationalisation fabriqué. Ça n'est pas du manque d'attention – c'est comme ça que le cerveau fonctionne. Pour accomplir des tâches de quelque complexité que ce soit, on a besoin de rendre automatiques les parties qui les composent – on n'atteindrait jamais le magasin si on avait à réfléchir à quel muscle de la cuisse bouger pour mettre la jambe dans la bonne position – et les comportements automatiques sont exactement des comportements auxquels on ne pense pas (c'est pour ça que les autobiographies de sportifs sont si ennuyeuses[10]).

Ainsi, non seulement vous posez une question à des gens à laquelle ils ne savent pas – et ne peuvent pas savoir – répondre, mais en plus vous leur demandez ça dans une belle salle de réunion, remplie d'autres personnes, après leur avoir donné des sous. Il n'est pas nécessaire de lire beaucoup en psychologie sociale pour comprendre que ça n'est pas exactement une situation idéale pour l'honnêteté. Les gens vont, évidemment, dire ce qu'ils pensent que vous souhaitez entendre, et même si vous faites poser les questions par le plus neutre des modérateurs, ils seront en mesure d'émettre une supposition éclairée sur ce que ça doit être.

C'est pour cela que regarder des groupes de travail est une expérience si exaspérante : comme une fille qui fait semblant de jouer l'idiote dans un bar, vous observez des gens se comporter de la façon qu'ils croient être ce qu'on attend des gens comme eux.

Mais si on ne peut ni mesurer les gens ni les interroger, que nous reste-t-il ? Eh bien, la bonne expérience à l'ancienne. Comme c'est souvent le cas dans la vie, il n'y a pas de raccourcis autour de l'incompétence ; à un moment, vous avez besoin de réelles aptitudes. Quant il s'agit de satisfaire les utilisateurs, il y a généralement deux parties : d'abord, il vous faut une dose de base d'empathie, la capacité à vous mettre à la place de l'utilisateur, et de voir les choses avec ses yeux. Mais pour que ça fonctionne, vous avez aussi besoin de savoir ce qu'il y a à l'intérieur de la tête de l'utilisateur, et d'après moi, la meilleure façon d'y arriver est simplement de passer beaucoup de temps avec eux.

10 Voir D. F. Wallace, « How Tracy Austin Broke My Heart » in *Consider the Lobster* (2005) [non-traduit en français].

Le meilleur expert en utilisabilité du monde, Matthew Paul Thomas, a commencé en travaillant quelques années en tant que support technique dans un cybercafé de Nouvelle-Zélande. C'est le genre de travail vers lequel on imaginerait Staline exiler des programmeurs, mais Thomas en a tiré le meilleur. Au lieu de devenir aigri à l'encontre des utilisateurs stupides qui ne comprennent pas ce qu'est une « barre des tâches », il en a eu ras-le-bol des idiots qui conçoivent des systèmes nécessitant ce genre de connaissances ésotériques. Et maintenant qu'il est en position de régler ce genre de problèmes, il comprend ces défauts de manière plus viscérale.

Je ne souhaite ce genre d'exil forcé à personne (même si je suppose que quelques concepteurs d'interfaces feraient de bons candidats), mais il n'y a certainement pas pénurie de gens qui possèdent déjà, à divers degrés, cette intuition de l'utilisateur. Le problème, c'est que personne ne les écoute. Il est toujours tellement facile de les taxer de naïveté, de stupidité ou de ringardise. Après tout, cette option de barre de menu additionnelle que vous souhaitez ajouter est parfaitement claire à vos yeux – quel mal pourrait-elle vraiment faire ?

Quand une entreprise se focalise sur ses usagers, c'est un vrai choc. Prenez Zappos, un magasin de chaussures en ligne. Chez Zappos, ce sont des fanatiques du service aux clients. Ils surclassent discrètement les primo-acheteurs en livraison « demain chez vous », ils écrivent des cartes et envoient des fleurs quand la situation le justifie, et ils ne se contentent pas de rembourser totalement les retours, mais payent aussi pour la réexpédition. C'est le genre d'entreprise que les gens couvrent d'éloges. Mais, et c'est plus intéressant encore pour notre sujet, s'ils n'ont pas en stock les chaussures que vous voulez, ils essayent de trouver un concurrent qui les a !

D'une perspective à court terme, ça parait absurde : pourquoi voudriez-vous travailler pour aider vos clients à acheter des chaussures chez quelqu'un d'autre ? Mais à long terme, c'est du génie. Bien sûr, vous ferez peut-être un achat quelque part ailleurs, mais non seulement vous reviendrez chez Zappos pour chaque achat de chaussures jusqu'à la fin de vos jours, mais vous allez aussi écrire des articles de blog longs et embrasés sur le fait que Zappo est incroyablement super.

C’est l’un des secrets du succès sur le Web : plus vous envoyez de gens ailleurs, plus ils reviennent. Le Web est plein de « nœuds feuilles » – des pages qui disent quelque chose d’intéressant, mais qui ne vous renvoient pas vraiment plus loin. Et les nœuds feuilles sont très bien – ils sont le cœur du Web, en fait – mais ce sont des fins de voyages, pas des commencements. Quand quelqu’un débute sa journée, ou démarre son navigateur, il veut une page qui l’emmène vers tout un bouquet de sites et de perspectives, pas qui essaye juste de le garder enfermé à un endroit. Quel est le site le plus populaire sur Internet, et de loin ? Google Search, un site dont le but est de vous envoyer ailleurs le plus rapidement et le plus discrètement possible.

La raison pour laquelle tous ces trucs à propos de pointes de métal, d’exil en Nouvelle-Zélande, de magasin de chaussures et de nœuds feuilles sont pertinents dans un livre sur les applications web est que je vais maintenant vous demander de faire quelque chose qui a l’air insensé, quelque chose dont on dirait que ça pourrait tuer votre site. Je vais vous demander d’ouvrir vos données. Abandonnez-les.

Je vous laisse une seconde pour reprendre votre respiration.

Ça n’est pas aussi fou que cela parait. Wikipedia, un site à succès quel que soit la façon de le mesurer, laisse les clés du magasin – vous pouvez télécharger l’intégralité des bases de données à un instant T, ce qui inclue en plus de chaque page de Wikipedia chaque changement fait sur chaque page, le tout avec la pleine autorisation de republier ça comme il vous plaira. Ça n’a pas l’air de nuire à sa popularité.

Bien évidemment, je ne suis pas en train de vous dire de publier à tous les vents les détails personnels de vos utilisateurs. Il serait fou de la part de Gmail de proposer un site où on pourrait télécharger chacun des mails de leurs utilisateurs. À la place, je vous suggère de permettre à vos utilisateurs de récupérer leurs propres données depuis votre site. Les gens qui importent leurs événements sur votre calendrier devraient pouvoir exporter leur calendrier ; les gens qui reçoivent leurs mails via Gmail devraient pouvoir les récupérer ensuite.

Permettre une bonne exportation n’est pas simplement la chose à faire parce que c’est juste, ça peut également être une façon efficace d’attirer des utilisateurs. Les

gens ne sont pas à l'aise avec l'idée de verser toute leur vie dans une application web hébergée – ils se sont trop faits avoir trop de fois par des entreprises qui leur ont pris toutes leurs données puis on fait faillite. Vous donner du mal pour vous assurer qu'ils puissent récupérer leurs affaires depuis votre site est un bon moyen de regagner leur confiance.

Bien que j'aie beaucoup de choses à dire sur les formats dans ce livre, la question du format que vous utiliserez ici n'est pas vraiment pertinente. Ce qui est important, c'est que vous le fassiez. XML, RDF, CSV – le système de blogs populaire Movable Type exporte simplement les articles sous forme de longs fichiers texte, et étant donné le format affreux qu'ils utilisent, c'est mieux que rien. À partir du moment où vous choisissez quelque chose d'à moitié sensé, les gens trouveront un moyen pour que ça marche.

L'exception c'est quand il y a déjà un standard dans votre domaine (de facto, ou pour une autre raison). Par exemple, OPML est très largement accepté comme la façon d'exporter la liste de blogs que vous lisez. Si c'est le cas, vous devez utilisez ce standard. Désolé. Si un autre logiciel fournit dans une mesure trop importante un certain format, vous devrez juste avaler la pilule et fournir une exportation dans ce format. Tout le reste semblerait grossier, et les utilisateurs ne vont pas s'intéresser aux détails techniques.

Et, bien sûr, ça va dans les deux sens : un bon moyen d'attirer les utilisateurs est de fournir une fonctionnalité d'importation vous-même. En permettant l'importation depuis d'autres produits, qu'ils aient des fonctionnalités d'exportation officielles ou non, vous rendez plus facile pour les utilisateurs la migration vers votre propre version. Même si vos concurrents n'ont pas de fonction d'exportation officielle, vous pouvez toujours aider les utilisateurs à sortir (et offenser vos concurrents) en grattant les données de leur système – en écrivant des outils maison pour récupérer les trucs depuis leur interface utilisateur et les verser dans votre base de données.

De tout ça résulterait le genre de monde sans friction dont les utilisateurs futés doivent rêver – glisser en douceur d'une application à l'autre, bénéficier des avantages d'une nouvelle fonctionnalité sans devoir abandonner ses anciennes données. Et si l'entreprise qui faisait ça se fait racheter, et que les développeurs qui

ont écrit toutes ces nouvelles fonctionnalités s'en vont et lancent un concurrent, vous pouvez récupérer vos données aussitôt et sauter sur la nouvelle application.

Ce qui signifie plus de choix – et n'est-ce pas ce qu'il y a de mieux pour tout le monde ?

CHAPITRE 5
CONSTRUIRE UNE PLATEFORME : FOURNIR DES APIS

L’autre semaine j’ai fait une de mes rares excursions hors de mon lit somptueusement attitré, et je me suis rendu à une fête locale. J’y ai rencontré un homme qui avait réalisé un site web pour charger et visualiser des données. Je lui ai demandé s’il avait une API, puisque cela semble vraiment utile pour un site tellement orienté vers les données. Il n’en avait pas, m’a-t-il dit ; ça demanderait trop de travail de maintenir à la fois une application normale et une API.

Je vous raconte cette histoire parce que ce type de la fête avait tort, mais probablement de la même façon que vous avez tort, et je ne veux pas que vous vous sentiez mal. Si même un jeune fondateur de start-up bien habillé dans un salon chic de Williamsburg commet cette erreur, ça n’est pas un grave péché.

En fait, l’erreur est que si vous construisez votre site web en suivant les principes exprimés dans ce livre, l’API n’est pas quelque chose de distinct de votre site normal, mais en est une extension naturelle. Tous les principes que nous avons évoqués – URLs intelligentes, GET et POST, etc. – s’appliquent aussi bien aux

sites web qu'aux APIs. La seule différence est qu'au lieu de renvoyer du HTML, vous voulez renvoyer du JSON à la place.

JSON (qu'on prononce « Jason »), pour les non-initiés, est un simple format pour échanger de bouts de données basiques entre logiciels. Basé originellement sur JavaScript, mais rapidement adopté par presque tous les langages principaux, il rend le partage de données sur le Web facile.

« Attendez ! », criez-vous peut-être, « je pensais que le XML servait à échanger des données sur le Web ». Malheureusement, vous avez été induit en erreur par une sinistre et néfaste campagne de relations publiques. XML est probablement quelque chose comme le pire format pour échanger des données. Voilà pourquoi :

Les langages modernes de programmation ont largement établi des standards concernant les mêmes composants de structure interne des données : entiers, chaînes, listes, sommes de contrôle, etc. JSON les reconnaît et rend facile le partage de ces structures de données. Vous voulez partager le nombre 5 ? Écrivez simplement 5. La chaîne « foo » est simplement "foo". Une liste des deux est simplement « [5, "foo"] » – et ainsi de suite.

C'est facile à écrire et à lire pour les humains, mais surtout, et c'est encore plus important, c'est automatique à écrire et à lire pour les ordinateurs. Dans la plupart des langages vous n'avez même pas besoin de penser au fait que vous utilisez JSON : vous demandez simplement à votre bibliothèque JSON de sérialiser une liste, et elle le fait. Vous lisez un fichier JSON, et pour votre programme c'est juste une structure normale de données.

XML, de son coté, ne supporte rien de tout ça. À la place, il pense en termes d'éléments avec des données textuelles, des instructions de programmation et des attributs, qui sont tous des chaînes. Publier des données en XML nécessite de comprendre comment faire rentrer au chausse-pied vos données internes dans un format spécifique, puis de vous assurer que vous l'avez décodé correctement. Encoder du XML est encore pire.

La raison principale pour laquelle XML est si mauvais pour échanger des données

est qu'il n'a pas été conçu pour ça en premier lieu. C'était un format pour baliser des documents textuels ; écrire des annotations avec des instructions de formatage de diverses sortes, à la manière d'HTML. C'est pour ça qu'il fait la distinction entre données textuelles et données d'attribut – les données d'attributs sont des choses qui ne font pas partie du texte véritable, comme :

```
J'attends avec impatience une démonstration <font color="green">couronnée de succès</font>.
```

Le mot « green » est une annotation, pas une partie du texte, donc va dans un attribut. Tout ça passe par la fenêtre quand vous commencez à parler de données :

```
<personne age="5">

<nom>Robert Booker</nom>

</personne>
```

Pourquoi « age » est un attribut, alors que « nom » est un élément ? C'est complètement arbitraire, parce que la distinction est absurde.

D'accord, donc XML a quelques fonctionnalités en plus dont personne n'a besoin. Quel mal à ça ? Eh bien, il manque aussi tout un paquet de fonctionnalités dont vous avez besoin – par défaut, XML ne supporte pas le concept on ne peut plus basique qu'est le nombre entier ; il n'y a que des chaînes. Et à cela s'ajoute le fait qu'il requiert d'utiliser le Schéma XML, une spécification tellement affligeante de complexité qu'elle verrouille mon navigateur quand j'essaye de l'ouvrir.

Mais le coût d'une telle complexité n'est pas simplement plus de travail pour le développeur – il se ressent véritablement sous la forme de bugs, particulièrement des failles de sécurité. Comme l'observe l'expert en sécurité Dan Bernstein, deux des principales sources de failles de sécurité sont la complexité (« Les failles de sécurité ne peuvent pas apparaître dans des fonctionnalités qui n'existent pas ») et l'encodage (« L'encodage est souvent source de bugs... Le décodage est souvent

source de bugs... Rares sont les occasions heureuses où l'encodage et le décodage font la même mauvaise interprétation de l'interface »).[11]

XML combine le pire des deux mondes : c'est un système incroyablement complexe à encoder. Sans surprise, XML a été responsable de centaines de failles de sécurité.[12]

À part le fait d'être plus simple, plus facile d'utilisation, plus sûr et plus rapide qu'XML, qu'est-ce que JSON a à offrir ? Eh bien, il a une fonctionnalité qui déchire qui lui a assuré cette victoire dans la guerre des formats : parce qu'il est basé sur JavaScript, il a une profonde compatibilité avec les navigateurs web.

Vous avez probablement entendu parler d'AJAX, une technique qui utilise la fonction XmlHttpRequest dans les navigateurs web modernes pour permettre aux pages de lancer leurs propres requêtes HTTP pour recevoir plus de données. Mais pour des raisons de sécurité, XmlHttpRequest n'autorise que des requêtes sur le même domaine que la page web d'où elles sont lancées. C'est à dire que si votre page est à l'adresse http://www.exemple.net/foo.html elle peut demander des choses comme http://www.exemple.net/info.xml, mais pas http://whitehouse.gov/data/dump.xml.

Pour les APIs, c'est une véritable catastrophe – tout l'enjeu de l'ouverture des données sur le Web réside dans le fait que les autres sites puissent les utiliser. Si vous êtes la seule personne qui puisse y accéder, pourquoi s'en donner la peine ?

Heureusement, il existe une exception : JavaScript. Une page web peut intégrer une balise HTML « <script> » qui pointe sur n'importe quel site sur Internet. Il y a même mieux : le code JavaScript peut ajouter arbitrairement ces balises script à la page. Le navigateur part alors chercher la page, et essaye de l'exécuter.

Maintenant, avec du JSON basique, ça ne serait pas très utile – la navigateur téléchargerait une liste, un objet, ou quelque chose, et ne saurait pas quoi en faire.

11 Voir http://cr.yp.to/qmail/guarantee.html.
12 Voir http://eve.mitre.org/.

Donc au lieu de simplement renvoyer du JSON, ça renvoie du JSON emballé dans un appel de fonction :

```
myCallback([5, "foo"]);
```

Et vous n'avez plus qu'à faire faire à la fonction « myCallback » ce que vous voulez qu'elle fasse avec les données.

Évidemment, si vous faites beaucoup de requêtes, vous voulez les maintenir séparées – elles ne peuvent pas toutes appeler myCallback. Alors vous supportez un paramètre de rappel qui vous permet de choisir le nom de la fonction. Ainsi, une URL comme http://www.exemple.net/info.json?callback=foo renverra :

```
foo([5, "foo"]);
```

La technique complète est connue sous le nom de JSONP et, naturellement, elle est automatisée par toutes les principales bibliothèques JSON, donc vous n'avez pas à vous embêter avec ces détails.

Très bien, donc maintenant nous disposons d'une pile de JSON, où est-ce qu'on la met. La réponse, bien sûr, est « à la même place que votre HTML ». Revenons à un exemple ancien, disons que nous avons des informations à propos d'un livre à l'adresse :

```
http://livres.exemple.org/b/3j7is
```

Où va le JSON ? Exactement au même endroit !

En fait, HTTP a une chouette fonctionnalité appelée Content Negotiation, négociation de contenu, qui permet à la même URL de renvoyer des formats différents selon l'émetteur de la requête. L'exemple classique de négociation de contenu est la transition depuis les images GIF vers le format d'images plus récent PNG. Certaines vieilles versions d'Internet Explorer ne supportaient pas le PNG ; les serveurs pouvaient utiliser la négociation de contenu pour leur envoyer les vieilles images GIF à la place.

La façon dont ça fonctionne est qu’à chaque fois que faites une requête HTTP (comme un GET), le client envoie en même temps une série d’en-têtes Accept, qui disent quels formats il aime. Voici un exemple typique :

```
Accept: text/html; q=1.0, text/*; q=0.8, image/gif ;
q=0.6, image/jpeg; q=0.6, image/*; q=0.5, */*; q=0.1
```

Ça dit que le navigateur préfère le HTML, puis prend le texte, puis les GIFs et les JPEGs, puis toutes les autres images, puis tout le reste.

Mais pour les APIs, nous n’avons pas besoin de faire quelque chose d’aussi compliqué. On peut simplement demander aux clients de notre API d’envoyer :

```
Accept: application/json
```

et faire en sorte que le serveur ouvre l’oeil là-dessus, et renvoie la version JSON s’il le voit. Dans le cas contraire, il renvoie la version HTML comme d’habitude.

Bien sûr, vous souhaiterez probablement proposer une option pour les gens qui ne ne sont pas à l’aise avec la négociation de contenu. Traditionnellement, on fait en sorte que :

```
http://livres.exemple.org/b/3j7is.html
```

force le serveur à renvoyer du HTML, tandis que :

```
http://livres.exemple.org/b/3j7is.json
```

renvoie toujours du JSON (on pourrait alors avoir :

```
http://livres.exemple.org/b/3j7is.json?callback=myCallback
```

pour supporter le JSONP).

Bon, soyons concrets. À quoi pourrait ressembler une de ces pages JSON ? Restons un moment sur notre exemple du livre. On peut imaginer une page sur un livre qui ressemblerait à quelque chose comme ça :


```
{

'id': '3j7is',

'titre' : 'The ABC book' ,

'mention_resp' : 'designed and cut on wood, by C. B. Falls. ' ,

'pagination': '30 p., ill. en coul.' ,

'description' : "Un chef-d'œuvre toutes époques confondues, et un classique dans son domaine, ce gros et bel abécédaire de l'artiste renommé C. B. Falls séduit et enchante les enfants depuis plus de quarante ans. M. Falls a conçu ce livre pour sa petite fille de trois ans, qui voulait un gros livre avec beaucoup d'images. Les illustrations sont gravées sur bois et imprimées depuis des plaques de quatre couleurs, et l'artiste a personnellement supervisé leur reproduction. L'imagination des petits et des grands est laissée libre de capturer le familier par ses propres moyens de reconnaissance, dans un médium nouveau et ancien où la couleur n'a pas obscurci le contour ni joué trop de tours à la nature.",

'editeur': 'Doubleday, Page & company',

'auteurs': [{'id': '0L115179A', 'nom': 'C. B. Falls'}],

}
```

Et si votre site autorise les gens à mettre à jour les pages sur les livres, vous pourriez imaginer supporter des requêtes PUT sur cette URI qui autorisent les gens à envoyer une nouvelle version de l'objet JSON. Vous l'encoderiez et effectueriez la mise à jour.

Ou, si vous laissez simplement les gens commenter les livres, vous pourriez les laisser envoyer en POST de simples données JSON vers la même URI où les commentaires sont envoyés d'habitude.

En fait, si vous le voulez vraiment, vous pourriez simplement les laisser envoyer en POST des données de formulaire et les encoder de la même façon que vous le feriez pour des données saisies dans un navigateur web. Puis vous pourriez leur indiquer le succès ou l'échec de la manœuvre par le biais des codes d'erreurs HTTP – une erreur 500 leur indiquerait un échec, alors qu'un code 303 See Other – une redirection vers la page de résultat de la requête – signifierait une réussite. Et quand ils suivraient la redirection et récupéreraient la page, celle-ci pourrait aussi négocier son contenu en JSON.

***

Très bien, il est temps maintenant d'aborder un sujet sensible. Je me suis retenu là-dessus, mais à un certain moment, cela devient inévitable. Oui, j'ai peur qu'il soit temps de parler de RDF.

Voyez-vous, tout ce truc JSON c'est formidable pour écrire des petits scripts sur des clients qui parlent à d'autres scripts sur des serveurs, mais ça laisse un peu à désirer quand on travail à l'échelle du Web. Il est difficile d'imaginer, par exemple, construire des outils vraiment utiles qui fonctionnent à travers différentes APIs JSON, à la façon des navigateurs web qui fonctionnent à travers des pages HTML de toutes sortes. Chaque API JSON a sa propre représentation interne, ses conventions et ses protocoles, ce qui implique qu'on doive écrire un code spécifique pour traiter avec chacune d'entre elles.

C'est là qu'intervient RDF. L'idée directrice est simple : et si on avait un format qui fait aux données ce que HTML fait aux documents – leur fournir une

représentation unique et cohérente qui supporte la nature hypertextuelle du Web. Ça n'est probablement pas clair du tout, alors regardons quelques exemples.

Les documents RDF sont très simples – ils sont composés de « triplets », des phrases simples avec trois parties : un sujet, un verbe (appelé prédicat), et un objet. Prenons un morceau de notre exemple plus haut, à savoir que le livre dont l'ID est 3j7is a pour titre « The ABC book » – en RDF, le sujet serait « 3j7is », le verbe « titre » et l'objet la chaîne « The ABC book ».

Seulement, RDF est pensé pour fonctionner à l'échelle du Web, donc au lieu de termes vagues comme « titre », tout a un URI. Comme dans :

```
<http://livres.exemple.org/b/3j7is#it>

http://www.w3.org/1999/02/22-rdf-syntax-ns#label

"The ABC book".
```

(Ces signes « # » sont là pour caractériser le fait que nous parlons du concept décrit dans la page web, et pas de la page elle-même.)

Bien sûr, taper toutes ces URLs à chaque fois fait vieillir vite, donc nous avons tendance à les abréger :

```
@prefix  rdfs:  <http://www.w3.org/1999/02/22-rdf-syntax-
ns#>.

<http://livres.exemple.org/b/3j7is#it>

rdfs: label

"The ABC book".
```

Voici une traduction rudimentaire du JSON d'au-dessus en RDF :

```
@prefix : <http://livres.exemple.org/api/schema#> .

<http://livres.exemple.org/b/3j7is#it>

:titre 'The ABC book';

:mention_resp 'designed and cut on wood, by C. B. Falls.';

:pagination: '30 p., ill. en coul.';

:description "Un chef-d’œuvre toutes époques confondues,
et un classique dans son domaine, ce gros et bel
abécédaire de l’artiste renommé C. B. Falls séduit et
enchante les enfants depuis plus de quarante ans. M. Falls
a conçu ce livre pour sa petite fille de trois ans, qui
voulait un gros livre avec beaucoup d’images. Les
illustrations sont gravées sur bois et imprimées depuis
des plaques de quatre couleurs, et l’artiste a
personnellement supervisé leur reproduction.
L’imagination des petits et des grands est laissée libre
de capturer le familier par ses propres moyens de
reconnaissance, dans un médium nouveau et ancien où la
couleur n’a pas obscurci le contour ni joué trop de tours
à la nature.";

:editeur 'Doubleday, Page & company';

:auteur <http://openlibrary.org/a/0L115179A#it>.

<http://openlibrary.org/a/OLl15179A#it>

:nom "C. B. Falls".
```

En plus d’utiliser invariablement des URIs, RDF a quelques fonctionnalités vraiment chouettes. En premier lieu, s’il y a quelque chose que l’on souhaite souvent faire avec des données, c’est de les combiner, et RDF rend ça très facile.

Pour combiner deux documents RDF, on se contente de les concaténer – ça n'est qu'une liste de faits ; deux listes de faits ensemble font une longue liste de faits. Ça n'est pas aussi simple en JSON, sans parler du XML.

Une autre belle fonctionnalité de RDF est que ça rend facile la concordance entre formats. La conversion entre deux formats JSON requiert tout le temps du code, mais avec RDF vous pouvez simplement publier un autre document RDF qui explique la table de concordance, comme :

```
rdfs:label = :titre.
```

De cette façon, un logiciel qui connaît « rdf:label » sait qu'il peut utiliser la propriété « :titre » de la même manière.

RDF a toutes ces fonctionnalités sympas, mais a un gros défaut : ça n'est en rien aussi simple à utiliser que JSON. Comme XML, ça a son propre modèle de données, ce qui implique d'écrire un code spécifique pour naviguer entre sa façon de voir le monde et la vôtre. Il existe quelques outils et techniques pour compenser cela (comme mon propre rdftramp(13), qui essaye de faire plus ressembler RDF a des objets Python normaux) mais ça reste un sérieux problème.

Le monde de RDF a essayé de remédier à cela en écrivant des solutions de remplacement RDF pour tous les outils existants dans monde des logiciels : des bases de données RDF, des langages de programmation RDF, des systèmes de requête RDF, des navigateurs RDF, des moteurs d'inférence RDF, et ainsi de suite. Si vous le voulez, il existe tout un monde de RDF dans lequel vous pouvez plonger.

En fin de compte, cependant, j'ai peur que ce ne soit pas une stratégie très prometteuse – ça va être difficile de créer des solutions de remplacement pour toutes ces choses qui soient aussi bonnes ou meilleures que les originales, et même si on y parvient, les gens garderont un attachement sentimental aux autres.

13 http://www.aaronsw.com/2002/rdftramp/.

Donc à ce point, je catégoriserais encore RDF comme une aspiration. Ça serait un beau format universel de publication – on pourrait faire beaucoup de choses avec – mais pour un travail de tous les jours, JSON est bien meilleur.

Ceci dit, RDF est bien sûr de très, très loin préférable à XML.

CHAPITRE 6

CONSTRUIRE UNE BASE DE DONNÉES : REQUÊTES ET SAUVEGARDES

Les APIs c'est bien et tout, mais elles sont plutôt limitées : elles ne donnent de réponses qu'aux questions qu'on sait déjà poser. Vous voulez en savoir plus sur le livre 3j7is ? Bien sûr, je vais tout vous dire. Mais vous voulez savoir quels livres publiés récemment partagent un auteur avec un livre publié il y a plus de cent ans ? C'est un peu plus compliqué.

Mais, heureusement, pas impossible. Il paraît ridicule de proposer votre propre API qui pourrait répondre à toutes sortes de questions de ce type. Mais vous vous souvenez de ces langages de requête dont on s'amusait dans le dernier chapitre ? Il s'avère que c'est exactement le genre de choses dans lequel ils excellent.

Le langage de requête RDF officiel s'appelle SPARQL (SPARQL Protocol And RDF Query Language, protocole et langage de requête RDF – qu'on prononce « sparkeul »). Si vous êtes familier de SQL, le langage standard de requête sur les bases de données, SPARQL vous semblera similaire, avec simplement RDF collé à tous les bons endroits. Voici comment on pose la question pré-citée en SPARQL :

```
PREFIX : <http://livres.exemple.org/api/schema#>

SELECT ?livrerecent, ?livreancien

WHERE {

?livrerecent :auteur ?auteur .

?livrerecent :annee_publication ?anneerecent .

FILTER ( ?anneerecent >= 2008 )

?livreancien :auteur ?auteur .

?livreancien :annee_publication ?anneeancien .

FILTER ( ?anneeancien <= 1908 )

}
```

Il y a beaucoup de choses ici, donc allons-y doucement. D'abord, on déclare simplement les préfixes de nos URIs, comme d'habitude. C'est juste un moyen de s'épargner de la saisie. Puis on dit qu'on veut se voir renvoyer les valeurs « ?livrerecent » et « ?livreancien ». En SPARQL, tout ce qui commence par « ? » est un substitut que le moteur de requête va chercher à remplacer avec quelque chose qui convient.

La clause « WHERE » annonce des contraintes à propos de ce qui convient pour remplacer ces substituts. « ?livrerecent » doit avoir un auteur et une année de publication, et cette année de publication doit être égale ou supérieure à 2008. « ?livreancien » doit aussi avoir un auteur et une année de publication – et, en outre, son auteur doit être le même que l'auteur de « ?livrerecent ». Mais son année de publication doit être égale ou inférieure à 1908.

Maintenant, comme SPARQL est conçu pour fonctionner à l'échelle du Web, vous n'avez pas besoin de garder cette requête à la maison. À la place, vous pouvez la pointer sur le système de recherche d'un autre serveur, qu'on appelle un SPARQL endpoint, ou une interface SPARQL. Vous n'avez peut-être pas beaucoup d'informations sur les livres, mais livres.exemple.org en revanche en a probablement – vous pouvez lui faire chercher des choses qui correspondent à votre requête.

Pour ce faire, vous prenez simplement la requête qu'on a généré au-dessus, et vous la collez dans une URL proprement formatée. Et – boum ! – en retour vous arrive une liste de réponses.

Et un autre aspect sympa de SPARQL est que, si on s'y prend bien, il permet de déployer les requêtes à travers plusieurs interfaces SPARQL. Ainsi, par exemple, on peut imaginer écrire une requête pour des livres dont les auteurs sont juifs. Les informations sur les livres et les auteurs peuvent être obtenues depuis le serveur livres, pendant que Wikipedia (dont la version RDF est appelée DBPedia) peut nous informer sur la religion des gens. Bien sûr, se débrouiller pour structurer ces requêtes d'une telle manière qu'elles ne prennent pas un temps infini est un projet de recherche en cours.

Dans le même temps, nous pouvons au moins aider ceux qui savent se débrouiller avec nos données et fournissant des sauvegardes en gros. La théorie, ici, est simple : il y a beaucoup de requêtes, de fusions, et de visualisations que les gens vont vouloir réaliser avec vos données, et qui sont impraticables avec quelque sorte d'API que ce soit, même aussi sophistiquée que SPARQL. Donc vous devriez tout aussi bien simplement leur donner une copie complète de votre jeu de données.

Et la pratique est encore plus simple : vous prenez simplement le JSON que vous générez pour chaque objet de votre site, et vous le mettez dans un unique gros fichier.

Il se peut que vous soyez tenté de le compresser, parce qu'il sera sans doute plutôt gros.

Chapitre 7
Construire pour la liberté : données ouvertes, sources ouvertes

Notre histoire commence avec un bourrage papier. On était en 1980 et le labo d'Intelligence Artificielle du MIT avait reçu une élégante nouvelle imprimante de chez Xerox. L'imprimante, cependant, avait une fâcheuse tendance à se bloquer, causant ainsi l'accumulation les tâches d'impression, et rien n'était imprimé avant que quelqu'un s'en aperçoive et supprime le bourrage.

Pour Richard Stallman, un des programmeurs du labo AI, ça n'était pas bien grave. Avec l'imprimante précédente, Stallman avait simplement modifié le pilote d'impression pour qu'il détecte les éventuels bourrages, et prévienne toutes les personnes qui lui avaient envoyé une tâche d'impression. « Si vous receviez ce message, vous ne pouviez pas présumer que quelqu'un d'autre s'en occuperait », raconte Stallman. « Vous deviez aller jusqu'à l'imprimante. Une minute ou deux après que l'imprimante soit tombée en panne, les deux ou trois personnes qui avaient reçu un message arrivaient pour réparer la machine. Généralement, sur ces deux ou trois personnes, une d'entre elles au moins saurait comment régler le problème. »

Mais l'imprimante Xerox était différente : Xerox n'avait pas fourni au labo le code source de leurs pilotes d'impression. Il n'y avait aucun moyen pour Stallman d'ajouter cette nouvelle fonctionnalité au pilote. Quand Stallman a demandé le code à Xerox, ils ont refusé de le communiquer, en insistant sur le fait que c'était un secret commercial important pour leur entreprise. Et quand Stallman trouva un étudiant de l'université de Carnegie-Mellon qui avait eu accès à ce logiciel, l'étudiant refusa également d'en fournir une copie, en disant qu'il s'était engagé par contrat avec Xerox à ne pas le partager.

Stallman était outragé. Les logiciels informatiques étaient sensés être des outils au service des gens ; c'est pour cela que lui et ses collègues de labo passaient leur temps à écrire des logiciels. Et là, par le biais d'une combinaison de cupidité et de restrictions légales, les gens étaient forcés de souffrir parce qu'on les empêchait d'améliorer ces outils.

Stallman voulut s'assurer que personne d'autre n'aurait à souffrir de cela ; il voulut construire un système d'ordinateur basé sur des principes de liberté. En 1984, il démissionna et annonça le projet GNU.

Stallman plus tard mit au clair que les logiciels libres étaient des logiciels qui garantissaient aux utilisateurs quatre libertés :

0. La liberté d'exécuter le programme, pour tous les usages.

1. La liberté d'étudier le fonctionnement du programme et de l'adapter à ses besoins.

(le code source est nécessaire pour cela.)

2. La liberté de redistribuer des copies du programme pour pouvoir aider son voisin.

3. La liberté d'améliorer le programme et de distribuer ces améliorations au public, pour en faire profiter toute la communauté.

(Encore une fois, le code source est nécessaire pour cela.)

« Je considère que la règle d'or requiert que si j'aime un programme je dois le partager avec d'autres personnes qui l'aiment. Je ne peux pas en bonne conscience signer un accord de non-divulgation ou un accord de licence de logiciel. Pour pouvoir continuer à utiliser des ordinateurs sans violer mes principes, j'ai décidé de réunir un corpus suffisant de logiciels libres qui me permette de m'en sortir sans aucun logiciel qui ne soit pas libre. »

Stallman a codifié ces libertés dans la licence publique générale GNU, ou GPL (General Public License). Si vous modifiez un bout d'un logiciel qui est sous licence GPL, et que vous le redistribuez, la licence implique que vous redistribuiez le code source gratuitement, et que vous autorisiez tous ceux qui reçoivent une copie à en faire de même.

Depuis 1984, le système d'exploitation GNU (dont le parfum le plus populaire est GNU/Linux) a été construit sous la licence GPL. Une étude de 2007 a montré que 13% des serveurs et 1% des ordinateurs de bureaux étaient vendus avec GNU/Linux. Et tout le monde peut télécharger gratuitement l'intégralité du système d'exploitation sur Internet.

Le succès de GNU/Linux a entraîne une montée en puissance du mouvement des logiciels libres, en même temps que le mouvement « open source », qui distribue des logiciels et leurs codes sources sous des licences de copyright qui apportent quelques unes des libertés du logiciel.

Le navigateur Mozilla Firefox, par exemple, est open source et représente en ce moment environ 15% du marché. De larges portions du système d'exploitation Mac OS X sont aussi en open source, dont Web Kit, le noyau de Safari, le navigateur de Mac OS X.

Les mouvements de l'open source et du logiciel libre ont aujourd'hui construit des alternatives libres pour à peu près tous les grands types d'applications informatiques, depuis le traitement de texte jusqu'aux jeux vidéos. Et pendant un moment, il a semblé que le rêve de Stallman était devenu réalité : on pouvait

véritablement continuer à utiliser les ordinateurs sans avoir de lois qui restreignent notre liberté – c'était possible « de s'en sortir sans aucun logiciel qui ne soit pas libre ».

***

Pendant ce temps, Tim Berners-Lee, un anglais vivant en France qui travaillait dans un labo de physique en Suisse, s'énervait devant les difficultés que rencontraient les physiciens pour partager des documents. Et donc, en 1989, il sortit le World Wide Web, développa les standards qui le font fonctionner, et fabriqua le premier navigateur web et le premier serveur web.

Le pouvoir du navigateur était sa flexibilité (ou, selon les termes du professeur de droit Jonathan Zittrain, sa « nature générative »). Tout comme un ordinateur classique vous permet d'utiliser n'importe quel programme, du lecteur audio à la calculatrice graphique, le navigateur web vous laisse voir n'importe quelle sorte de document. Un livre, un article de physique, ou des photos de chats avec des légendes rigolotes – le navigateur web s'en fiche ; il affiche tout ce que le serveur lui fournit.

Ça semble être un détail trivial aujourd'hui, mais c'était un grand changement par rapport aux autres logiciels en réseau de l'époque. Les programmes d'emails, par exemple, sont conçus pour simplement afficher des emails – ils ont une interface énormément spécialisée pour composer des emails, trouver des emails, voir les émetteurs et les destinataires des emails, et classer des emails dans des dossiers différents. C'était vrai aussi des logiciels de discussion, des logiciels de chat, et des autres bouts de logiciels qui communiquaient sur le réseau.

Le Web était différent : il n'était pas spécialisé dans un type particulier de contenu, mais vous laissait partager ce que vous vouliez.

L'absence de spécialisation du navigateur web a permis aux gens de déplacer cette spécialisation vers le serveur web. Le serveur web traditionnel servait simplement des documents statiques que quelqu'un avait écrit préalablement. Mais il s'avéra rapidement clair qu'il n'y avait aucune raison que le serveur soit si limité.

Au lieu de simplement servir des documents préalablement composés, le serveur pouvait composer des documents « à la volée » dès qu'ils étaient demandés. Ainsi, au lieu d'avoir seulement un document qui liste les restaurants qui ont des tables disponibles, on pouvait charger le serveur de contacter les différents restaurants, d'apprendre leurs disponibilités, et de construire une page avec les résultats.

Et les utilisateurs, au lieu de demander passivement des documents pré-écrits, pouvaient soumettre des requêtes au serveur et commencer réellement à interagir avec lui. Ainsi, ils pouvaient demander au serveur de réserver une des tables, et envoyer leurs noms et numéros de téléphone avec cette requête.

Il en résulta que l'humble navigateur web commença rapidement à supplanter toutes les autres applications « spécialisées ». Au lieu d'avoir un programme spécial pour simplement lire des emails, les gens lisent leurs emails sur le Web. De la même manière, les groupes de discussion, les salons de chat, et les autres formes d'interaction sociale se sont déplacées dans le navigateur web.

Mais les développeurs de logiciels ont vite découvert que, pour les créatures sociales comme nous les humains, tout a un composant d'interaction sociale. Par exemple, donner un titre et catégoriser les photos que vous prenez pourrait sembler être une activité manifestement solitaire. Pourtant, des sites comme Flickr ont démontré que les gens adorent commenter et catégoriser les photos de leurs amis, ou même d'étrangers, et que les gens, tout bien considéré, préféreraient organiser leurs photos dans un programme qui les expose à d'autres personnes.

Le résultat, c'est le phénomène récent « Web 2.0 », par lequel à peu près tous les morceaux de l'informatique ont migré dans le Web et été rendus sociaux d'une manière ou d'une autre. Pour les photos et les vidéos, il y a Flickr et Youtube. Pour les infos, il y a des sites comme Digg et Reddit où on peut soumettre, éditer, et voter pour des informations. Les calendriers, les listes de tâches, même les collections musicales et les traitements de texte se sont tous transformés en applications web dynamiques et sociales.

Les experts discutent maintenant d'un futur pas si lointain de « clients idiots » et de « cloud computing » où les autres applications de l'ordinateur disparaîtraient et

tout ce qui resterait serait le navigateur web. Et pour les gens qui utilisent des ordinateurs en libre-accès ou les cybercafés, ce futur est déjà là.

Pour certains, c'est une perspective excitante. Mais pour ceux, comme Stallman, concernés par les problèmes de la liberté logicielle, c'est effrayant. Même dans les jours sombres du pilote d'impression propriétaire, Stallman avait encore le contrôle sur l'ordinateur qui pilotait l'imprimante, même s'il n'avait pas le code source pour le modifier. Mais avec une application Web 2.0, vous n'avez même pas ça. L'ordinateur qui fait tourner votre logiciel est enfermé dans une ferme de serveurs loin d'ici. Vous pouvez seulement communiquer avec lui à travers votre navigateur web.

Bon, cela apporte une certaine flexibilité. Les navigateurs web peuvent être programmés pour bloquer les publicités ou pour extraire du contenu. Des modules comme Zotero et Greasemonkey permettent aux utilisateurs d'ajouter des fonctionnalités aux sites existants en interceptant et en modifiant les documents quand ils arrivent du serveur web.

Mais c'est plutôt une pâle notion de la liberté, comme si on disait que les cinéphiles ont le contrôle sur les films qu'ils voient parce qu'ils peuvent tenir des images devant l'écran pendant qu'ils les regardent.

Une autre option, bien sûr, est de fournir des APIs. Ainsi, au lieu d'avoir à cliquer manuellement sur le bouton « Acheter » d'une page Amazon pour acheter un nouveau lot de rasoirs, avec une API Amazon, vous pouvez avoir un programme qui achète automatiquement les rasoirs pour vous chaque mois.

C'est sans aucun doute pratique, mais encore une fois, c'est une bien pâle notion de liberté comparée aux quatre libertés apportées par le logiciel libre. Si Amazon était vraiment libre, vous ne seriez pas simplement capable d'automatiser votre usage de l'application, vous seriez capable de changer comment l'application fonctionne véritablement.

La solution évidente à ce défi est simplement de diffuser le logiciel sur le serveur Web sous la licence GPL ou sous une autre licence de logiciel libre. Ainsi, tout le monde pourrait télécharger une copie et la modifier tout son soûl. Et une nouvelle

version de la GPL est sortie, l'AGPLv3, qui requiert que les personnes qui utilisent le logiciel dans leur application web rendent leur logiciel librement disponible pour les utilisateurs de l'application.

Mais seule une application complètement asociale consiste purement en un logiciel. La grande majorité d'entre elles sont intéressantes parce qu'elles vous donnent accès à des données auxquelles ont aussi contribué d'autres utilisateurs. Par exemple, le logiciel qui laisse les gens éditer des pages web est sans doute la chose la moins intéressante concernant Wikipedia. La raison pour laquelle ce site est si populaire, c'est parce que beaucoup de gens ont accumulé leur savoir dans ce logiciel.

Wikipedia s'est attelé à ça en allant un pas plus loin – non seulement le code source est libre, mais les données le sont aussi. N'importe qui peut télécharger une copie de la base de données de Wikipedia (à l'exception des informations personnelles des utilisateurs) et commencer sa propre imitation de Wikipedia basée sur l'originale. Et on peut modifier sa copie du logiciel de Wikipedia pour en adapter le fonctionnement à ses propres goûts.

C'est beau en théorie, mais en pratique, bien sûr, personne ne le fait. Même si votre version de Wikipedia était pleine de fantastiques nouvelles fonctionnalités, il serait tout de même presque impossible de trouver quelqu'un pour l'utiliser. Les gens utilisent Wikipedia parce que c'est là que tous les autres sont ; il est pratiquement impossible de faire bouger tout le monde.

Pour Wikipedia, le problème est quelque peu atténué par le fait d'avoir une sorte de contrôle pseudo-démocratique sur le site. Wikipedia est piloté par un bureau élu par (une faible fraction de) ses utilisateurs, et le bureau a un pouvoir nominal de contrôle sur le logiciel et les modifications qui y sont faites. Mais ça ne ressemble encore que de loin à la liberté que les utilisateurs de GNU/Linux ont dans le monde non-connecté. Présenter sa candidature, être élu, puis soutenir vos modifications devant une bureaucratie rétive au changement est beaucoup plus compliqué que modifier quelques fichiers du code source sur son ordinateur puis redémarrer.

Et donc, les partisans les plus acharnés de la liberté logicielle proposent que nous renvoyions le balancier depuis l'informatique sur un serveur centralisé vers son monde de départ où nous faisions tous tourner nos applications sur nos machines locales. Seulement cette fois, au lieu d'être des applications qui n'utilisent pas le réseau ou qui ne parlent qu'à un serveur distant, ce seraient des applications pair-à-pair, peer-to-peer, qui chercheraient d'autres utilisateurs et qui interagiraient directement avec eux.

De grandes enjambées ont été faites dans la construction de logiciels pair-à-pair, en raison, pour une part non négligeable, de l'intérêt massif pour l'utilisation de cette technologie dans le partage de musique sans se faire attraper par les représentants de la loi. Mais, particulièrement quand on la compare à la technologie centrée sur les serveurs du Web 2.0, la technologie pair-à-pair en est encore à ses balbutiements. Écrire une application sociale qui serait pair-à-pair est à peu près mille fois plus compliqué qu'écrire le même programme en tant qu'application web.

Pourtant, les logiciels pair-à-pair, si on arrivait à les faire fonctionner, pourraient présenter le meilleur des deux mondes : la liberté de modifier comment le programme fonctionne sur notre ordinateur local, aussi bien que la possibilité de partager et de collaborer avec d'autres sur Internet. Et donc, pour ceux qui se soucient de liberté (comme pour ceux qui se soucient d'échanger de la musique), ça semble être une voie importante à approfondir.

Dans le même temps, même si, à l'instar de la question de comment formuler des requêtes sur de nombreuses bases de données SPARQL, le problème de la liberté des applications web reste irrésolu, vous pouvez toujours commencer. L'Open Knowledge Foundation, un groupe qui promeut les bases de données librement partagées, a proposé une définition du service logiciel ouvert(14). La définition codifie essentiellement les principes évoqués plus haut :

1. Rendez votre code disponible en tant que logiciel libre ou open source.

2. Rendez vos données disponibles en tant que connaissances ouvertes.

14 http://opendefinition.org/software-service/.

Pour les logiciels libres / open source, il y a la liste officielle de l'Open Source Initiative et de la Free Software Foundation pour dire si votre licence est suffisamment libre ou ouverte. Les exemples incluent la licence Expat/BSD, la GPL, etc. L'Open Knowledge Foundation, similairement, liste une série de licences, incluant certaines licences Creative Commons, la licence GNU Free Documentation, et ainsi de suite.

C'est l'aspect légal, mais l'aspect technique est tout aussi simple : fournissez un entrepôt de code source pour tout votre code, et des sauvegardes SQL pour toutes vos données.

Bien sûr, cela laisse de nombreuses questions ouvertes. Que faire des données personnelles, par exemple ? Mon sentiment personnel est que chacun devrait être autorisé à télécharger ses propres données et toutes les données qui lui sont accessibles par le biais de l'interface web – par exemple, les données concernant leurs amis Facebook.

Et il y a plein de place pour expérimenter la conception de sites qui promeuvent plus de contrôle démocratique. Peut-être que vous pourriez essayer des choses et m'en parler, et ils les ajouteront à la seconde édition de ce livre.

CHAPITRE 8
CONCLUSION : UN WEB SÉMANTIQUE ?

Bon, nous avons traversé pas mal de choses ensemble, vous et moi. Nous avons construit une application depuis ses humbles URLs jusqu'à ses pompeuses prétentions démocratiques. Le long du chemin, nous avons rendu son monde sûr pour les robots, les systèmes de requêtes, les chercheurs, et Richard Stallman. Mais comment passer de cette sorte de page web à la grande vision de Web sémantique dont on a tant entendu parler ?

Commençons par réaliser que rien que d'être sur le Web est quelque chose de fascinant. Les URLs fournissent un schéma unifié d'adressage pour tous les documents – ce qui est plutôt miraculeux. Imaginez essayer d'expliquer aux gens d'antan ces mots incroyables que vous pouvez donner à quelqu'un, qui peut les amener chez lui, les entrer dans une boite qu'il a sur son bureau, et obtenir exactement l'article, l'image ou la vidéo que vous évoquiez. Pour une génération pour qui la récupération d'information c'était se rendre à la bibliothèque locale, remplir quelques formulaires, et attendre quelques semaines pendant qu'ils essayent de comprendre ce que vous cherchez et de mettre la main dessus, ça représente un sacré changement.

Mais REST va encore plus loin. En rendant les documents accessibles par les moteurs de recherche grâce à un protocole standard, vous n'avez même plus besoin de connaître la bonne URL de ce que vous cherchez. Tapez simplement quelques mots choisis dans Google et boum ! en retour vous arrive la chose que vous vouliez, en un quart de seconde.

Bien sûr, ça n'est pas que Google ; REST rend possible une tapisserie interconnectée qui supporte tout, des navigateurs web aux éditeurs web, jusqu'aux proxies intermédiaires de traduction.

Un acte difficile à suivre. Et ensuite vint la possibilité d'obtenir non seulement des documents depuis ces serveurs lointains, mais aussi des données. En important et exportant des données brutes, nous avons rendu possible de se passer des programmes, en instaurant un marché à peu près libre de produits en compétition, ce qui crée un surplus massif de consommateurs. Allez nous !

Et nous ne nous sommes pas arrêtés là. Se contenter de laisser les utilisateurs emporter leurs données à la maison, c'est de la petite bière comparé au partage des données avec le monde entier (imaginez un monde entier de gens chez eux avec leur petite bière de données). C'est pourquoi le Web a inventé les APIs, qui nous permettent de partager des données avec tous ceux qui pensent pouvoir en faire usage.

Là, nous ne sommes plus un simple site web, qui envoie des pages au navigateur, en fournissant une liste de type « voir aussi » présentant d'autres pages qu'on pourrait visiter. On échange les données elles-mêmes d'application à application, rendant possible un monde nouveau de mélanges et d'applications intelligentes.

C'est un concept entièrement nouveau de tapisserie – une tapisserie de données et non plus de documents. Les documents ne peuvent pas vraiment être fusionnés, intégrés, et recherchés ; ils servent plutôt d'instances isolées destinées à être vues et révisées. Mais les données sont protéiformes, capables d'endosser la forme la plus à même de répondre à vos besoins.

Mais avec la variété de nos besoin qui s'élargit, nous avons besoin d'un meilleur

moyen pour parvenir aux données qui nous seront les plus utiles. C'est là qu'interviennent nos requêtes et nos sauvegardes. Nous ne sommes plus entravés par le fait d'être seulement capables de poser les questions que les programmeurs d'un site avaient prévues et prises en compte ; maintenant nous pouvons poser les questions que nous voulons quelles qu'elles soient, ou accomplir des procédures qui ne peuvent même pas être formulées comme des questions. En combinant ces sauvegardes de différentes sources de données, les possibilités sont infinies.

Mais à partir de là, où allons-nous ?

Assurément, la première étape est de prendre les volumineuses sauvegardes que nous avons faites, et de les charger dans une unique grosse base de données. Et, bien sûr, on a commencé à voir des gens faire ça, depuis les projets de recherches jusqu'aux compagnies commerciales comme Metaweb avec Freebase. Freebase est une énorme base de données collaborative éditable sur le Web de type RDF, pré-remplie de données extraites de Wikipedia et de nombreuses autres sources, et complétée avec les contributions d'utilisateurs divers. Freebase est encore plutôt petite, mais leurs objectifs sont ambitieux – créer une base de données qui combine de multiples sources différentes et la mettre à disposition en tant que support pour des gens qui veulent construire des applications plus intelligentes.

Dans l'idéal, bien entendu, les applications intelligentes ne seront pas dépendantes d'un seul site commercial, comme Freebase, mais fusionneront et combineront les connaissances issues de divers sites disséminés sur le Web, en se déplaçant et en ramassant des informations plus utiles, et en décidant quelles parts de celle-ci elles peuvent croire.

Déjà, nous pouvons voir des choses de ce genre dans des projets de recherche. Un des outils du Web sémantique les plus excitants est un programme appelé cwm, bricolé et assemblé entre deux réunions (ou durant celles-ci) par Sir Tim Berners-Lee en personne. Cwm (qu'on prononce coum) est un des programmes les plus étonnants que j'aie vus ; c'est un véritable couteau-suisse pour les données, entièrement conçu autour de RDF.

Bien sûr, il remplit toutes les tâches de base – lire et écrire des fichiers RDF dans divers

formats, combiner plusieurs fichiers, afficher tous les résultats dans un joli format.

Naturellement, il peut aussi chercher dans les données obtenues pour répondre à vos question d'une façon très similaire à celle de SPARQL.

Mais cwm va un pas plus loin. Il ne se contente pas de chercher dans les données ; il réfléchit à leur propos. Cwm peut suivre des règles logiques ; prenez celle-ci, par exemple :

```
{ ?x a :Man } => { ?x :mortality : mortal } .
```

(Si quelque chose est un homme, alors il est mortel.) Chargez cela dans cwm, en même temps que :

```
: Socrates a :Man .
```

(Socrate est un homme.) Et il va logiquement en déduire que Socrate est mortel. De telles séries de règles ont évidemment toutes sortes d'utilisations possibles, depuis les jeux de société logiques jusqu'à la véritable programmation. Mais une utilité manifeste est de fournir des conversions entre différents formats RDF. Imaginez deux schémas : « joe: », qui a « petit », « moyen », et « grand » et « starbucks: », qui a « short », « tall », « grande », et « venti ». Maintenant, nous pouvons simplement écrire quelques règles pour les convertir entre eux :

```
{ ?x joe:taille joe:petit } <=> { ?x Starbucks:taille
Starbucks:tall } .

{ ?x joe:taille joe:moyen } <=> { ?x Starbucks:taille
Starbucks:grande } .

{ ?x joe:taille joe:grand } <=> { ?x Starbucks:taille
Starbucks:venti } .
```

Chargez ces règles dans cwm, avec les données dans un format, et cwm va « réfléchir » là-dessus et recracher les données dans l'autre format.

Mais cwm peut faire beaucoup plus que de la déduction logique de base. Il a aussi une grande variété de fonctionnalités intégrées, qui peuvent tout faire du traitement mathématique à la cryptographie avancée. Avec elles, et quelques règles intelligentes, vous pouvez même écrire des programme entiers en utilisant cwm.

Accessoirement, rien de tout ça n’est neuf – à peu près tous ces trucs étaient écrits en 2001.

Cwm est également assez malin pour aller sur le Web et trouver plus de règles comme celles-ci, en farfouillant à travers les pages web à la recherche de quelques morceaux de RDF qui le rendraient plus malin encore. Il peut suivre les liens et les URLs et en récupérer plus de données, en surfant sur le Web comme un adolescent qui s’ennuie surfe sur le Web pour se distraire.

Si ça vous intéresse d’essayer ce procédé par vous-même, vous pouvez tester le dernier projet de Tim : le Tabulator. C’est une petite extension pour votre navigateur web qui lui permet de voir des documents RDF en plus des pages web normales. Soudain, les documents ne sont plus de simples listes de balises ennuyeuses ou de texte, mais un sentier de liens cliquables que vous pouvez suivre au gré de vos envies (et, avec les versions ultérieures, vous pouvez même éditer certains des champs).

On peut imaginer des outils comme cwm et Tabulator installés derrière les applications que nous utilisons tous les jours, et les enrichissant des connaissances tirées des étendues du Web.

Parce que c’est la véritable idée derrière le Web sémantique : laisser les logiciels utiliser l’immense génie collectif intégré dans les pages qui y sont publiées. Pensez à tous les cas où les logiciels utilisent des APIs ou des bases de données : votre correcteur d’orthographe interroge un site web pour trouver la définition d’un mot, votre carnet d’adresse cherche à savoir si vos amis sont en ligne, votre agenda télécharge une page pour vous faire suivre un événement à venir. Maintenant, imaginez si ces programmes n’étaient pas limités à un site en particulier, mais pouvaient aspirer l’intelligence d’Internet en liberté.

Votre correcteur d'orthographe peut suggérer des mots liés ou alternatifs, ou simplement se maintenir au courant de l'argot moderne. Votre carnet d'adresse peut vous dire où sont vos amis dans l'instant, et ce qu'ils ont fait dernièrement. Votre agenda peut garder un œil sur les événements qui pourraient peut-être vous intéresser.

C'est facile de se moquer de ce genre de visions. Mon père, après avoir vu ce genre de démonstration, demandait toujours « Mais pourquoi ton grille-pain a besoin de suivre la bourse ? ». Et, si ça se trouve, ça n'en vaut pas la peine. Mais le Web sémantique est fondé sur un pari, le pari que donner au monde des outils pour collaborer et communiquer facilement va ouvrir des perspectives si formidables que nous ne pouvons qu'à peine les imaginer aujourd'hui.

Bien sûr, ça a l'air un peu dingue. Mais ça a payé la dernière fois qu'ils s'y sont risqué : on a fini avec un petit truc appelé le World Wide Web. Voyons s'ils peuvent encore le faire.

Texte mis en page sur Scribus

octobre 2014

Cette œuvre courte est le premier jet d’un manuscrit d’Aaron Swartz, rédigé pour la collection « Synthesis Lectures on the Semantic Web » à l’invitation de son directeur, James Hendler. Malheureusement, le livre n’a pas été terminé avant le décès d’Aaron en janvier 2013. En son hommage, le directeur et l’éditeur publient l’œuvre numérique gratuitement.

Extrait de l’introduction de l’auteur :

« ... nous commencerons par essayer de comprendre l’architecture du Web – ce qui marche bien et, à l’occasion, ce qui ne marche pas, mais surtout pourquoi il est fait comme ça. Nous allons apprendre comment il autorise à la fois les utilisateurs et les moteurs de recherche à coexister pacifiquement tout en supportant tout, du partage de photos aux transactions financières.

Nous continuerons en réfléchissant à ce que signifie construire un programme au sommet du Web – comment écrire un logiciel qui sert équitablement autant son utilisateur immédiat que les développeurs qui veulent construire par-dessus lui. Trop souvent, une API est boulonnée au dessus d’une application existante, après-coup ou comme un morceau complètement différent. Mais, comme nous le verrons, quand une application web est conçue proprement, les APIs en découlent naturellement et leur maintenance ne requiert que peu d’efforts.

Puis nous nous intéresserons à ce que signifie pour votre application de n’être pas seulement un autre outil pour les gens et les logiciels, mais une partie de l’écologie – une section du Web programmable. Ce qui implique d’exposer vos données à être interrogées, et copiées, et intégrées, et ce même sans autorisation explicite, au sein du plus grand écosystème de logiciels, tout en protégeant la liberté des utilisateurs.

Enfin, nous conclurons avec une discussion sur cette expression très galvaudée : « Web sémantique », et nous tâcherons de comprendre ce qu’elle signifie vraiment. »